Elcor's Machine Description System: Version 3.0

Shail Aditya, Vinod Kathail, B. Ramakrishna Rau
Information Technology Center

HPL-98-128
October 1998

Email: {aditya, kathail, rau}@hpl.hp.com

retargetable compilers,
table-driven compilers,
machine description,
processor description,
instruction-level parallelism,
EPIC processors,
VLIW processors,
HPL-PD,
EPIC compilers,
VLIW compilers,
code generation,
scheduling,
register allocation

This report is intended to serve as a user and reference manual for Elcor's Machine Description System. In this report, we address the information that Elcor, our re-targetable EPIC compiler for HPL-PD, needs about the processor for which it is generating code. This information makes it possible to write Elcor as a "table-driven" compiler that has no detailed assumptions regarding the processor built into the code. Instead, it makes queries to a machine-description database which provides the required information about the processor. Consequently, Elcor can be retargeted to different HPL-PD processors by merely changing the contents of this database, despite the fact that the processors vary quite significantly. This report describes the internal data structures of Elcor's machine description database, the procedural interface between the compiler and the database, and a high-level data-description language for specifying processors within the HPL-PD space.

This report supersedes all documents related to Version 2.0 of Elcor's Machine Description System.

1 Introduction

The first-generation of VLIW processors[1] were motivated by the specific goal of cleaning up the architecture of the attached processor sufficiently to make it possible to write good compilers [1, 2]. Such processors typically had higher levels of instruction-level parallelism (ILP), more registers, and a relatively regular interconnect between the registers and the functional units. Furthermore, the operations were RISC-like (in that their sources and destinations were registers), not "micro-operations" (which merely source or sink their operands from or to buses). Recently, with increasing levels of integration, DSPs have begun to appear that have a VLIW architecture [3, 4]. This current generation of embedded VLIW processors reflects the state of the art of VLIW in the mini-supercomputer space a decade ago [5, 6].

In the meantime, VLIW has continued to evolve into an increasingly general-purpose architecture, providing high levels of ILP and incorporating a number of advanced features [7, 8]. This evolved style of VLIW is termed *Explicitly Parallel Instruction Computing (EPIC)*. HPL-PD architecture [8] defines the space of EPIC processors which are of interest to us in this report. In the rest of this report, we shall use the term EPIC to include VLIW as well.

For EPIC processors, since the primary focus is on achieving high levels of ILP, the most important compiler task is to achieve as short a schedule as possible. The scheduler and register allocator are, therefore, the key modules. These two topics have received a great deal of attention over the years in many research communities, resulting in a vast body of literature. In this report, we are concerned not with the scheduling and register allocation algorithms but with the information that Elcor, our re-targetable EPIC compiler for HPL-PD, needs about the processor for which it is performing these functions. The identification of this information makes it possible to write "table-driven" EPIC compilers that have no detailed assumptions regarding the processor built into the code. Instead, such a compiler makes queries to a *machine-description database (mdes)* which provides the required information about the processor. Such a compiler can be retargeted to different EPIC processors by changing the contents of this database, despite the fact that the processors

[1] Note that our use of the term VLIW processor is specifically intended to differentiate it from the array processors of the past and the DSPs of today. Even though these might have certain VLIW attributes, such as the ability to issue multiple operations in one instruction, we view them as being half way between VLIW processors and horizontally microprogrammed processors.

vary widely, including in the number of functional units, their pipeline structure, their latencies, the set of opcodes that each functional unit can execute, the number of register files, the number of registers per register file, their accessibility from the various functional units and the busing structure between the register files and the functional units.

An mdes-driven compiler is of particular value in the context of developing a capability to automatically synthesize custom EPIC ASIPs (Application-Specific Instruction-Set Processors), where one of the obstacles, both to the evaluation of candidate designs as well as to the use of the selected one, is that of automatically generating a high-quality compiler for the synthesized processor.

The mdes infrastructure in Elcor is shown in Figure 1. In order to make the compiler fully parameterized with respect to the target machine information, we separate the modules that need the information from the machine-description database where this information is stored. The compiler modules are only allowed to make a fixed set of queries to the database through an *mdes query system (mQS) interface*. Any form of database organization could be used for this purpose as long as the interface is well defined. However, specializing the internal structure of the database to expedite the more frequent queries results in substantial performance improvements.

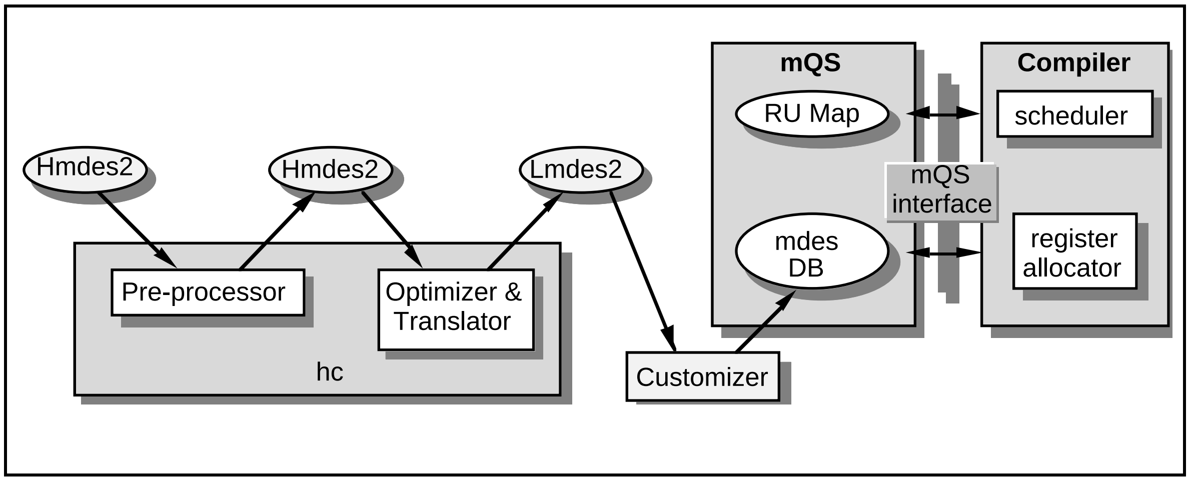


Figure 1: The mdes infrastructure in Elcor.

In order to specify the machine description information externally in a textual form, the Elcor compiler uses the high-level machine description database specification Hmdes2 [9]. This external format is organized as a general relational database description language

(DBL) which supports a high-level, human-editable textual form and a low-level machine-readable form. Tools are provided to expand macros in the high-level form and to compile it down to the low-level form. The exact definitions of the various database relations used by a database are also defined within the same framework and serve to specialize the file format for a given database schema.

As shown in Figure 1, the machine-description for a given target architecture is expressed in Hmdes2 format as a text file. After macro processing and compilation, the corrresponding low-level specification, expressed in Lmdes2, is loaded into Elcor using a *customizer* module that reads the specification and builds the internal data structures of the mdes database.

In this report, we discuss the mdes-driven aspects of compilers for EPIC processors. It is important to stress that our goal is not to be able to build an mdes-driven compiler that can target any arbitrary processor. The target space is limited to a stylized class of EPIC processors for which we know how to generate good code using systematic rather than ad hoc techniques. This space is described is Section 2.

Section 3 articulates our model of phase ordered EPIC code generation. It also summarizes the concepts of binding hierarchies for operations, opcodes and registers as well as the Operation Binding Lattice (OBL). A more detailed description appears in a companion report [10].

In Section 4, we discuss the key information about the target machine that has to be stored in the mdes and how it is used by an EPIC compiler. Our focus in this report is restricted to the mdes required by the scheduler and register allocator, since these are the most important modules of an EPIC compiler. Currently, we do not address other phases of the compiler such as code selection, partitioning across multiple clusters, optimizations concerned with the memory hierarchy, and final code assembly.

Whereas Section 4 discusses the underlying concepts, the following three sections are intended to be closer in spirit to reference manuals.

- Section 5 describes the internal data structures of Elcor's machine description database (*mdes*). This is for the benefit of those who might wish to add new queries to the mQS interface and who will, therefore, require an understanding of the manner in which the machine description information is represented in the mdes.

- Section 6 defines the procedural interface between the compiler and the mdes, which is called the mdes Query System (*mQS*). This is for the benefit of compiler writers who wish to use the existing information present within the mdes database.
- Section 7 describes the high-level language Hmdes2 for describing processors within the HPL-PD space and creating an mdes for them. This is for the benefit of those who would like to experiment with new processors or modify existing ones.

Lastly, Section 8 reviews the antecedents of this work, as well as other related work.

2 The space of target EPIC processors

HPL-PD and Elcor have been designed to address a certain space of EPIC processors that provide relatively high levels of ILP. This space of processors was consciously defined in such a way as to promote the implementation of efficient, high-quality compilers. Figure 2 shows an example EPIC processor in this space which we shall use throughout this report to illustrate various points. We identify the characteristic features of our target space below.

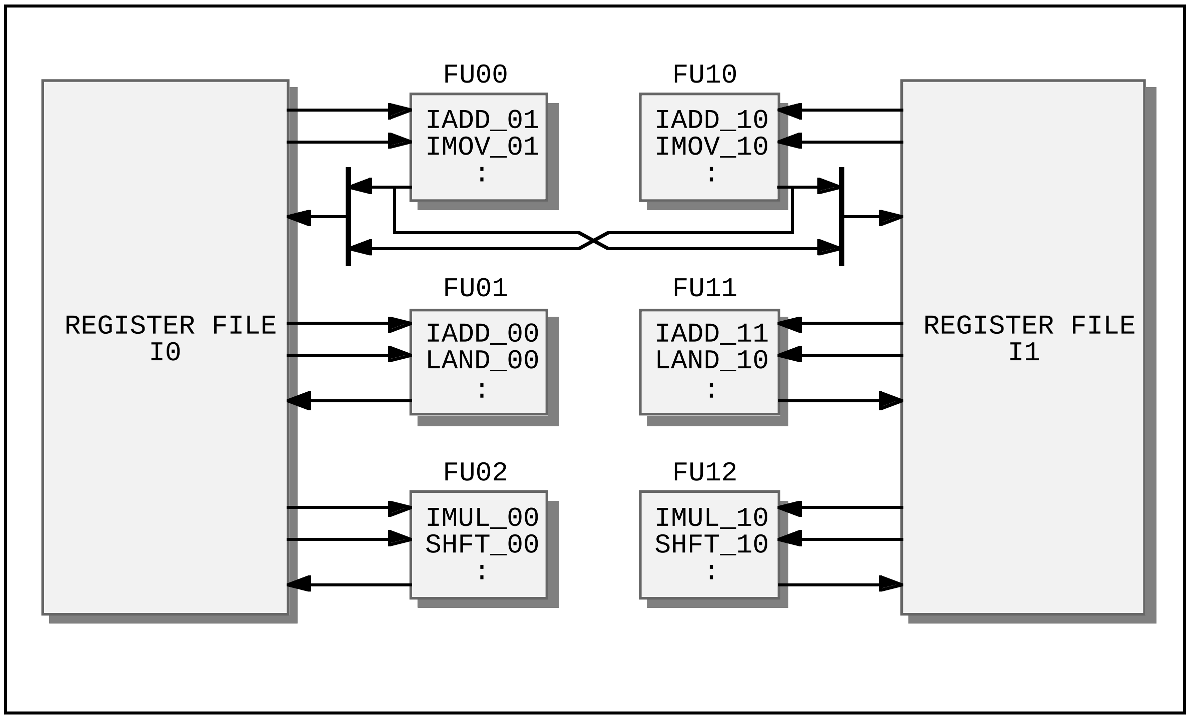


Figure 2: A sample EPIC processor.

- **Multiple pipelined functional units**. Since the EPIC architecture was developed to exploit ILP, the most basic characteristic of our space of target EPIC processors is the presence of multiple functional units (FUs) that can execute multiple operations in parallel. Often, the parallelism is such as to allow multiple identical operations to be issued simultaneously. In Figure 2, integer add operations (IADD) can be performed on four of the FUs, whereas integer multiply operations can be performed on the other two FUs. The machine has been designed in such a way that all six FUs can initiate operations simultaneously.

- **MultiOp**. A characteristic of EPIC processors is that the available ILP is specified explicitly via an instruction format that permits the specification of multiple operations per instruction (*MultiOp*). All operations in a single instruction may be issued in parallel without the hardware having to check the correctness of doing so. It is assumed that the requisite checking was performed by the compiler before placing those operations into the same instruction.

- **Non-unit assumed latencies (*NUAL*)**. In real machines, there are always at least a few opcodes that take more than one cycle to generate their results. Also, depending on the relative importance of such an opcode, its execution may be not pipelined at all, be fully pipelined (i.e., a new operation can be issued every cycle), or be partially pipelined (e.g., a divide may take 24 cycles to complete, but a new divide can be started every 10 cycles). RISC and superscalar processors usually provide a sequential program interface in which the assumed latency of every operation (or, in this case, every instruction) is one. This permits each operation to use the result of a previous operation as one of its input operands. If, in reality, the previous operation has a latency greater than one, and is directly followed by a dependent operation, the processor must ensure that the correct value is, nevertheless, carried from the first operation to the second operation (for instance, by using register interlocks). Such execution semantics are referred to as *unit assumed latencies (UAL)*.

  EPIC processors expose architecturally visible latencies via execution semantics that we call *non-unit assumed latencies (NUAL)*. With, NUAL visible latencies are no longer constrained to be one. NUAL allows the dependence checking responsibility to be eliminated from hardware and instead performed by a compiler which ensures that assumed latencies are satisfied.

- **Register-register operations**. A fundamental requirement for facilitating the task of writing a scheduler is that it be possible to write all results, produced during the same cycle, to a register file. This relieves the scheduler from the difficult constraint of scheduling all consumers of a data item exactly at the time it is produced, literally "reading off" the item from the result bus. In an EPIC processor, all of the operations other than loads and stores operate from register to register. Optionally, there can be register bypass logic that provides the additional capability to pass the result directly from a result bus to a source bus, but this merely reduces the effective latency of the pipeline.

- **Register files.** Rather than a heterogeneous collection of registers with differing special capabilities and interconnectivity, the registers in an EPIC processor are organized into one or more register files, each of which consists of a number of similar registers.
- **Large numbers of registers per register file.** Rather than taking the view that most of the operands reside in memory (and that the machine provides the requisite memory bandwidth), the view taken is that most operands will be in registers. The expectation is that a variable will primarily be referenced in a register with, perhaps, an initial load from memory or a final store to memory. Consequently, a relatively large numbers of registers are provided. The code generator takes advantage of this and is designed with the view that register spill is infrequent, not the norm[2]. This also reduces the memory bandwidth requirement, which is particularly valuable at high levels of ILP.

Modern EPIC processors have a number of additional features that we also include within our space of target architectures.

- **Heterogeneous functional units.** When designing a processor that can issue multiple integer operations per cycle, it is unusual for all of the integer or floating-point FUs to be replicas of one another. In Figure 2, the processor has been designed to be able to perform an IADD on every one of four FUs. But on two of them, it can perform an IMOV (copy) operation, while on the other two it can perform an LAND (logical-AND) operation. This leads to a set of heterogeneous FUs which, in the extreme case, are all dissimilar.
- **Multiple register files.** From the compiler's viewpoint, the most convenient arrangement is for all similar registers to be in the same register file, e.g., all the 32-bit registers in one file and all the 64-bit registers in another one. But in a highly parallel

[2] This assumption is central to our use of the phase ordering described in Section 3.2.

EPIC processor, this implies very highly ported register files, which tend to be slow and expensive. Instead, such highly ported register files are split into two or more register files, with a subset of the FUs having access to each register file. In our example processor in Figure 2, the integer registers have been divided into two register files, I0 and I1.

If a processor is such that for any given operand of any given operation (e.g. integer add) there is only one register file that can serve as the source or destination, we call it a *single-cluster* processor. This does not necessarily mean that there is just one register file. For instance, in Figure 2, if the three functional units on the right were floating-point units, then this would be termed as a single-cluster processor since the operands of all integer opcodes would have to be in register file I0, whereas the operands of all floating-point opcodes would have to be in I1. In this case, there is no ambiguity as to where the operands can be placed. On the other hand, a processor such as that shown in Figure 2, is termed as a *multi-cluster* processor since every operation in a computation can be executed on either the functional units on the left or on those on the right. In each case, its operands would need to reside in distinct register files. As a result, in this machine the some opcodes implementing the same operation (e.g. IADD_01 and IADD_10) are not interchangeable while others (e.g. IADD_01 and IADD_00) are.

- **Shared register ports.** It is often impractical to provide every FU with dedicated register ports for every opcode that it can execute. Opcodes that are provided for completeness, but which are infrequently used, use register ports that they share with other FUs. FU00 and FU10 each need a write port into both register files to permit data to be copied from one register file to the other. Instead of dedicated ports, they share a single write port per register file. Thus, while F00 is writing the destination register of an IMOV_01 into I1, FU10 cannot be completing an operation with destination I1 at the same time. Correct code generation must honor this constraint.
- **Shared instruction fields.** Likewise, bits in the instruction word are a valuable commodity, and must often be shared. The instruction format that permits the specification of a long literal may do so by using the instruction bits that would normally have specified, for instance, the third operand of a multiply-add operation. This precludes the multiply-add opcode, but not the two-input multiply or add opcode, from being issued in the same cycle as the opcode that specifies a long literal.
- **Interruption model.** The schedule constructed by the compiler for a program can be disrupted by events, such as interrupts, page faults or arithmetic exceptions, which

require that the execution of the program be suspended and then be resumed after the interrupt or exception handler has been executed. We shall refer to such events, generically, as *interruptions*. We shall consider two hardware strategies for dealing with interruptions. The first one halts the issue of any further instructions, takes a snapshot of the processor state for the executing program, including the state of execution of the pipelined operations, and then invokes the interruption handler. After the handler has executed, the program's processor state is restored, and program execution is resumed. The net effect, from the viewpoint of the program, is that its execution was frozen in place during the execution of the interruption handler. We refer to this as the *freeze model*.

The second strategy halts the issue of any further instructions, but permits those that have already been issued to go to completion. Once the pipelines have drained, the interruption handler is executed, after which instruction issue picks up where it left off. We refer to this as the *drain model*[3]. Although the drain model is considerably less complex to implement from a hardware viewpoint, it requires that specific measures be taken during scheduling and code generation to ensure correctness. These measures are discussed in Section 4.5.

In the case of pipelined branch operations, i.e., branches which have a latency of more than one cycle between the initiation of the branch operation and the initiation of the target operation, we always use the freeze model regardless of the model used for the rest of the operations. Since the semantics of a branch are to alter the flow of control and start issuing a new stream of instructions, the drain model, which would permit the branch to continue to completion, but which would nevertheless prevent any further instructions being issued, is meaningless.

In addition to those mentioned above, EPIC processors can have a number of other features such as predication, control speculation, data speculation, rotating registers and programmatic cache management [7, 8] which, individually and collectively, can have a great impact on the quality of the schedule. Since these features do not introduce any additional issues that are pertinent to the subject of this report, we do not discuss them here.

[3] In previous works by the authors, the freeze and drain models have been termed as the EQ model and the LEQ model, respectively. We believe that that way in which we conceptualized these issues in the past was imprecise, and that our current terminology better reflects our current way of thinking about exception handling.

3 Code generation for EPIC processors: an overview

In this section, we summarize the conceptual background required to understand and motivate the rest of this report. A much more detailed description and motivation for the concepts presented here appears in the companion report [10].

3.1 The compiler's view of a processor

Conceptually, the compiler is playing the role of a human assembly language programmer. As such, it is not interested in the structural description of the hardware. More precisely, it is only interested in the hardware structure indirectly, to the extent that this is reflected in the architectural description of the machine as might be found in an architecture manual. Specifically, rather than needing the equivalent of Figure 2 in the form of an HDL description, the compiler only needs to know, for each opcode, which registers can be accessed as each of its source and destination operands[4]. Table 1 lists some of the opcodes and the set of registers[5] to which each of the source or destination operands of those opcodes can be bound.

An operation consists of an *opcode* and a *register tuple*—one register each per operand. An important point to note is that the choice of register for one operand can sometimes restrict the choice for another operand. For instance, imagine that in our example processor, FU00, in addition to the two read ports from I0, also possessed its own read port from I1 which could be used to source either of the inputs of FU00. In this case, IADD_01 could source a single register in either I0 or I1 as its left or right input. But it cannot simultaneously source two registers in I1 since it has only one read port from it. Thus the accessibility of registers by an opcode must be specified in terms of which register tuples are legal. It is insufficient to merely specify which registers are accessible on an operand by operand basis. Consequently, the first row in Table 1 should be interpreted as stating that every register tuple in the Cartesian product $\{I0\} \times \{I0\} \times \{I0\}$ is legal for the opcode IADD_00.

---

[4] Additionally, for an EPIC processor, the compiler also needs to know the relevant latencies and resource usage of each of these operations, since it is the responsibility of the compiler to ensure a correct schedule. We shall return to this issue later on.

[5] As a notational convenience we shall use {I0} and {I1} to refer to the set of all the registers in register files I0 and I1, respectively.

Note that Table 1 does not contain any explicit information about the structure of the hardware. Instead, for each opcode, it specifies which sets of registers can be the sources or the destination. The compiler does not need to know about FUs, register files and interconnects; all it needs to know is which opcodes can access which registers. This opcode- and register-centric view will be reflected throughout the report.

Table 1: The input-output behavior of selected opcodes

Semantics	Opcode	Source 1	Source 2	Destination
Integer Add	IADD_00	{I0}	{I0}	{I0}
	IADD_01	{I0}	{I0}	{I0,I1}
	IADD_10	{I1}	{I1}	{I0,I1}
	IADD_11	{I1}	{I1}	{I1}
Integer Multiply	IMUL_00	{I0}	{I0}	{I0}
	IMUL_10	{I1}	{I1}	{I1}
Copy	IMOV_01	{I0}	-	{I0,I1}
	IMOV_10	{I1}	-	{I0,I1}

Thus, the mdes must serve two needs: firstly, to assist in the process of binding the operators and variables of the source program to machine operations by presenting an abstract view of the underlying machine connectivity (subject of the rest of this Section) and, secondly, to provide the information associated with each operation needed by the various phases of the compiler (discussed in Section 4).

3.2 A model of ILP code generation

For reasons of compile-time efficiency, the code generation process for an EPIC processor must use a phased approach in mapping the source program's operations to the processor's architectural operations [10]. These phases are:

1. code selection
2. pre-pass operation binding (including partitioning, if needed)
3. scheduling
4. register allocation and spill code insertion

5. post-pass scheduling
6. code emission

These phases are discussed further in Section 3.2.4. Each phase successively refines and narrows down the options available for either the opcodes, the registers, or both, finally yielding architectural operations that can be executed by the processor. Optimization phases may be inserted at various points in this sequence but, since they do not affect the level of binding, we ignore them. This section defines the various levels of operation, opcode and register hierarchies in the context of the above phase ordering. These hierarchies also form the basis for organizing the mdes information to be used by the above phases.

3.2.1 Code generation: semantic operations to architectural operations

Semantic operations and architectural operations form the two end points of the EPIC code generation process. In this section, we describe the properties of these two types of operations that are relevant to our discussion in the subsequent sections.

Semantic operations. These are the operations in the input to the code generation process, and they correspond to actions performed by a pre-defined virtual machine. The opcodes at this stage of compilation are the *semantic opcodes* provided by the pre-defined virtual machine. An operand is either a constant (such as 3 or 4.1) or a *virtual register* (VR) or an element of an *expanded virtual register* (EVR) [11]. Although EVRs are not part of any language, some of the machine-independent loop-transformations preceding the code generation may introduce them into the code. EVRs either can be mapped to rotating registers [6, 8] or can be converted to simple VRs by code transformation (e.g., loop-unrolling) [12].

Architectural operations. These are the operations in the output of the code generation process, and they represent commands performed by the target machine. The opcodes at this stage are *architectural opcodes* available in the target machine. To simplify the exposition, we treat any addressing modes supported by the target machine as part of the opcode rather than part of an operand. An operand is an *architectural register* which is either a literal or a register in the target machine. For uniformity, we model a literal as the contents of a read-only "literal register". Thus an architectural register is either a real machine register or a literal register.

Between semantic and architecture operations, we define a lattice of operation bindings (Section 3.2.4) that represents refinement and implementation decisions taken during the various phases of the code generator. In order to do that, we first need to define the various levels of register and opcode refinements that are used at various points in this lattice.

3.2.2 The register binding hierarchy

Compiler-registers. A *compiler-register* is either a single architectural register or a set of architectural registers, with a fixed spatial relationship, that are viewed as a single entity by the compiler.

Compiler-registers sit just above the architectural registers and provide a convenient abstraction for compiler and OS conventions. For example, some architectures view an even-odd pair of single-precision floating-point registers as a double-precision compiler-register. Also, most phases of the compiler can work with compiler-register abstractions such as the stack pointer, the frame pointer, and parameter passing registers without worrying about the specific architectural registers reserved for these purposes; only the register allocator need be aware of the exact correspondence.

Note that the translation from compiler-registers to architectural registers is primarily a book-keeping step as there are no decisions to be made.

Generic register sets. A *generic register set* is a maximal set of compiler-registers that have the same storage attributes (as defined in Section 4.3).

Generic register sets provide the first layer in translating operands in the semantic layer, i.e., VRs and constants, to architectural registers. They provide a useful abstraction, since they focus on the properties that are relevant to the abstract computation model and ignore details such as the physical connectivity of the target machine.

Access-equivalent register sets. In order to help us define the middle layer of the register hierarchy, which is called an *access-equivalent register set*, we first define a few other terms. An *alternative* is a triple consisting of a compiler-opcode (as defined in Section 3.2.3), a latency descriptor and a reservation table[6], that are jointly valid for the target processor. A *register set tuple (RS-tuple)* is a tuple of register sets, such that each register set is a subset of a single generic register set (i.e., all the registers have the same storage attributes). An *access-equivalent RS-tuple* corresponding to a given alternative is a maximal

[6] Latency descriptors and reservation tables are discussed in Section 3.3.4. For now, it suffices to say that a latency descriptor provides all the information needed by the scheduler regarding an operation's latencies, and a reservation table describes its resource usage over time.

RS-tuple, where each register set corresponds to one of the operands of the compiler-opcode, and every register tuple in the Cartesian product of the register sets is jointly valid with that alternative, taking into account both the connectivity constraints of the processor as well as the instruction format constraints. Each register set in an access-equivalent RS-tuple is an access-equivalent register set. As an example, Table 2 shows all of the (maximal) access-equivalent RS-tuples for the various operations in our example processor.

Table 2: Access-equivalent RS-tuples and opcode sets in the context of our phase ordering. ({I0} and {I1} each represent a set of registers.)

Semantics	**Opcode**	**Source 1**	**Source 2**	**Destination**
Integer Add	{IADD_00,IADD_01}	{I0}	{I0}	{I0}
	{IADD_01}	{I0}	{I0}	{I1}
	{IADD_10}	{I1}	{I1}	{I1}
	{IADD_10,IADD_11}	{I1}	{I1}	{I1}
Integer Multiply	{IMUL_00}	{I0}	{I0}	{I0}
	{IMUL_10}	{I1}	{I1}	{I1}
Copy	{IMOV_01}	{I0}	-	{I0}
	{IMOV_01}	{I0}	-	{I1}
	{IMOV_10}	{I1}	-	{I1}
	{IMOV_10}	{I1}	-	{I0}

The significance of defining this middle layer in the register hierarchy is the following. Each access-equivalent register set contains registers that are interchangeable with respect to that opcode after scheduling has taken place; any register can be used in place of any other without any impact on the correctness of a scheduled piece of code. Furthermore, since all the register tuples implied by an access-equivalent RS-tuple are architecturally valid, the compiler-register for each operand can be independently selected by the register allocator.

Since all compiler-registers in an access-equivalent set have identical storage attributes, we can also associate these attributes with the set itself. The hardware notion of a register file is an example of an access-equivalent register set.

3.2.3 The opcode binding hierarchy

Compiler-opcodes. In some cases, architectural opcodes do not provide the right abstraction that the compiler, especially the scheduler, needs. For example, on a given architecture, move operations may be implemented by adding 0 or multiplying by 1, and a 64-bit add may be implemented using four 16-bit adds. But the compiler may not want to expose these facts. Thus, we introduce an abstraction over architectural opcodes, called *compiler-opcodes*. A compiler-opcode is implemented by one or more architectural opcodes as specified by a *bundle-macro*, which is described in detail in Section 4.2. As with registers, the translation from compiler-opcodes to architectural opcodes is essentially a book-keeping step; there are no decisions to be made.

Generic opcode sets. A *generic opcode set* is the maximal set of compiler-opcodes that implement the same function, e.g., integer add.

These sets provide the first layer in translating the semantic opcodes to architectural opcodes and provide a useful abstraction during the code selection phase by allowing one to focus on the semantics of the operations available on the target machine and hiding details such as the connectivity of functional units to register files.

Access-equivalent opcode sets. These sets are defined using the notion of access-equivalent register sets introduced in the last section. An *access-equivalent opcode set* is the maximal set of compiler-opcodes that are part of the same generic opcode set (i.e., implement the same function) and for each of which there is an alternative that yields the same access-equivalent RS-tuple. The access-equivalent opcode sets in our example processor are shown in Table 2.

Note that this definition permits each compiler-opcode to access the access-equivalent RS-tuple with a different latency descriptor or reservation table, but it does require that the RS-tuple be equally accessible by every compiler-opcode in the opcode set. Therefore, these compiler-opcodes can be interchanged for each other without having to insert or delete any copy operations. This gives the scheduler maximal freedom in scheduling without changing the computation graph.

3.2.4 The operation binding lattice (OBL)

Since an operation is composed of an opcode and one or more operands, we can describe all possible ways of binding semantic operations to architectural operations as a binding matrix obtained by taking the cross-product of the steps used in binding opcodes with the steps used in binding registers. Figure 3 shows the operation binding matrix. The columns correspond to the various levels in the opcode binding hierarchy described in Section 3.2.3. The first four columns correspond to the levels in the opcode hierarchy within the compiler, the last column represents the architectural opcodes in the target machine. The rows in the matrix correspond to the levels in the register binding hierarchy described in Section 3.2.2. The first four rows correspond to the levels in the register binding hierarchy used within the compiler, the last row represents the architectural registers in the target machine.

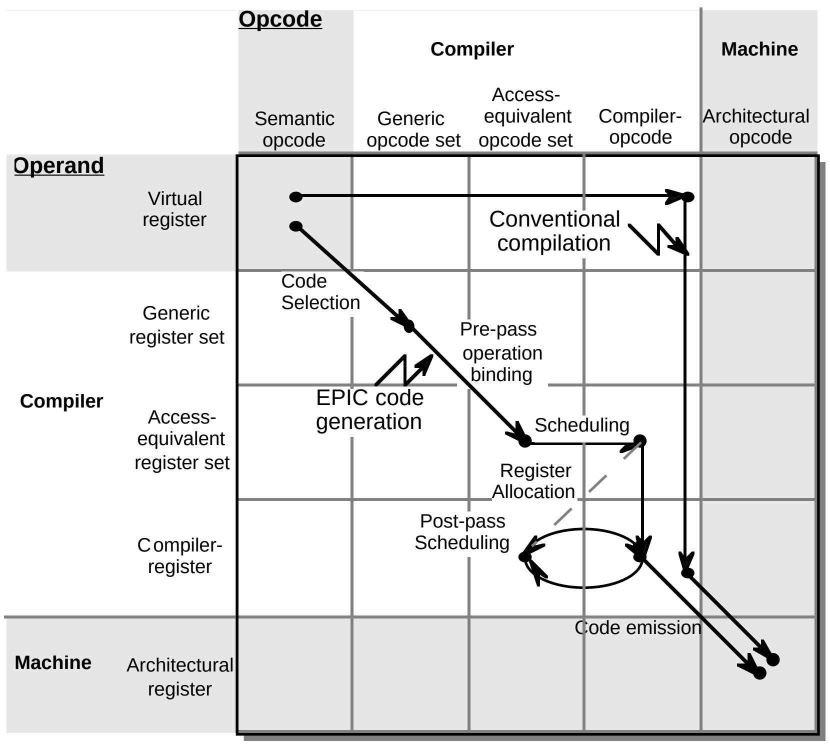


Figure 3: The operation binding matrix.

A left to right walk of a row in the matrix represents the steps used in binding semantic opcodes to architectural opcodes but keeping the operand binding fixed. Similarly, a top to bottom walk of a column represents the steps used in binding virtual registers to architectural registers but keeping the opcode binding fixed. Some compiler modules may refine the binding of both opcodes and operands simultaneously; this corresponds to moving diagonally in the matrix. Note that, in general, a binding decision made for either one (opcode or register) may restrict the choices available for the other. To simplify the exposition, our discussion as well as the matrix in Figure 3 assume that all operands in a program follow the same binding path. Although this is typically the case, exceptions may exist. For example, some compilers may bind constants to literal registers during code selection, and virtual registers to real architectural registers in the register allocation phase.

The top-left corner and the bottom-right corner in the matrix correspond to the two ends of the binding spectrum, and any path that connects these two end points defines a phase ordering that can be used to structure the code generation process.

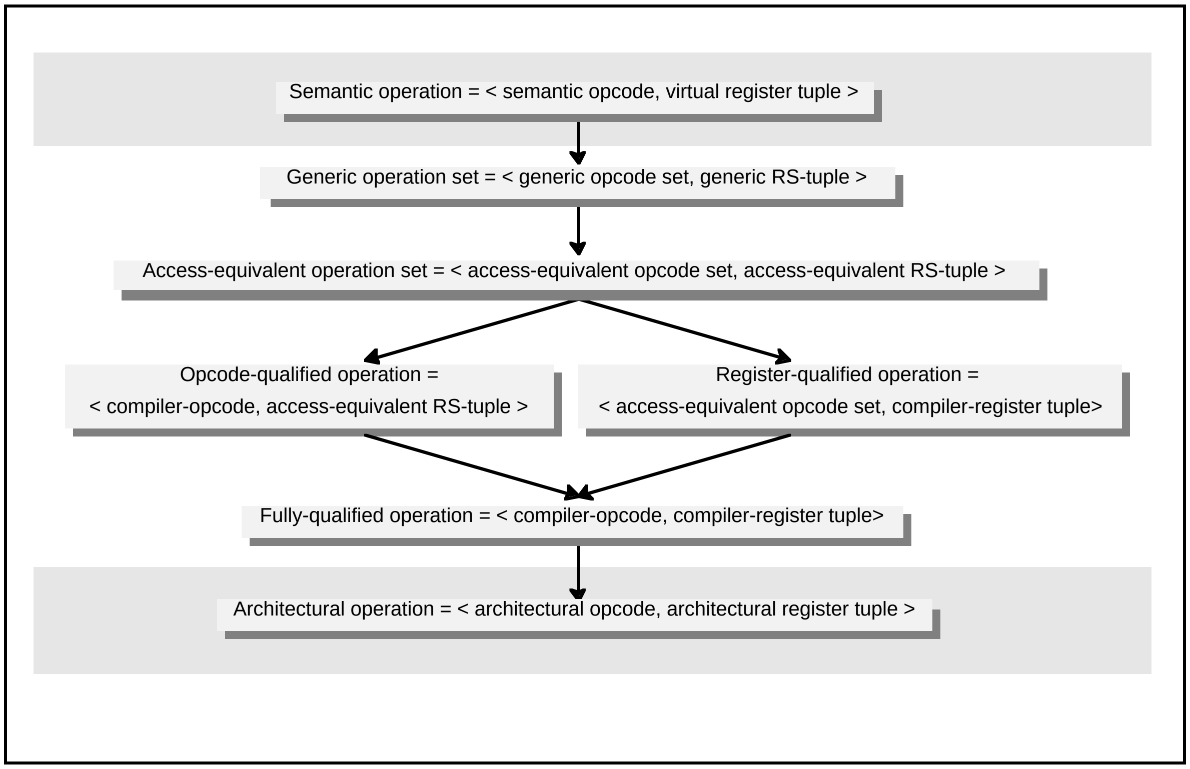

Semantic operation = < semantic opcode, virtual register tuple >
Generic operation set = < generic opcode set, generic RS-tuple >
Access-equivalent operation set = < access-equivalent opcode set, access-equivalent RS-tuple >
Opcode-qualified operation = < compiler-opcode, access-equivalent RS-tuple >
Register-qualified operation = < access-equivalent opcode set, compiler-register tuple>
Fully-qualified operation = < compiler-opcode, compiler-register tuple>
Architectural operation = < architectural opcode, architectural register tuple >

Figure 4: Operation binding lattice for our preferred phase ordering. Semantic and architectural operations are not part of the binding lattice. They are included to show implementation relationships; that is, semantic operations are implemented using generic operation sets, and fully-qualified operations are implemented using architectural operations.

Compilers for traditional RISC architectures follow the path labeled "conventional compilation". Semantic opcodes are mapped to the compiler abstraction of architectural opcodes during the code selection phase. Scheduling of operations simply spreads them apart in time to honor latency constraints. Then the register allocation phase maps virtual registers to the compiler abstraction of architectural registers. Finally, the code emission phase converts the code to assembly code for the target machine.

In contrast to traditional RISC architectures, the EPIC code generation process involves many more steps. In Figure 3, the path labeled "EPIC code generation" shows our preferred phase ordering for EPIC code generation. The preferred phase ordering induces a lattice of operation bindings, which we call the *operation binding lattice*, as shown in Figure 4. Note that semantic and architectural operations are shown in the figure, but they are not part of the lattice. They are used to show "implementation relationships"; semantic operations are implemented by generic operation sets and architectural operations implement fully-qualified operations. We briefly describe the various phases as they step through the operation binding lattice.

Code selection. The code selection phase maps semantic operations to *generic operation sets*, i.e., it maps semantic opcodes and virtual registers to generic opcode sets and generic register sets, respectively. Note that the mapping from semantic opcodes to generic opcodes is not, in general, one-to-one. A further point to note is that a phase, such as the code selection phase, may have internal sub-phases that correspond to moving horizontally or vertically in the binding matrix.

Pre-pass operation binding. At this point, the generic operation sets may contain multiple *access-equivalent operation sets*, each consisting of an access-equivalent opcode set along with its access-equivalent RS-tuple. Such operations need to be further bound down to a single access-equivalent operation set. This is done by the pre-pass operation binding phase. The constraint that must be satisfied is that each operation in the computation graph has to be annotated with an access-equivalent operation set in such a way that, for every variable, the intersection of the access-equivalent register sets imposed upon it by all of the operations that access it, called its *access-equivalent register option set*, must be non-empty. A variety of heuristics, which are beyond the scope of this discussion, may be employed in doing so. In the case of a multi-cluster processor, one of these is a partitioning algorithm which attempts to distribute the given computation over the clusters. This may introduce copy operations which move data between register files in order to get a balanced distribution. Since the access-equivalent opcode and register sets are closely interrelated, the pre-pass operation binding phase partially binds both opcodes and registers simultaneously. For our example processor, each integer add, multiply or copy operation would be bound to one of the access-equivalent operation sets listed in Table 2.

Scheduling. The scheduling phase is one of the main phases of an EPIC code generator. For each operation, the scheduler decides the time at which the operation is to be initiated. It also determines which compiler-opcode is to be used as well as the reservation table and latency descriptor that are used by the operation, i.e., it picks a specific alternative. In the case of statically scheduled EPIC machines, the scheduling phase refines access-equivalent operation sets to *opcode-qualified operation sets*, i.e., operations in which the possible alternatives have been narrowed down to a particular one, as a consequence of which the opcode options have been narrowed down to a single compiler-opcode, but with the register options having been narrowed down only to an access-equivalent RS-tuple[7].

Register allocation. The register allocation phase assigns a specific compiler-register to each of the virtual registers in the computation graph by selecting one of the compiler-registers from the corresponding access-equivalent register set. This yields *fully-qualified operations*, i.e., a specific alternative and a specific compiler-register tuple.

The register allocation phase may introduce additional code to spill registers to memory. The spill code is fully-bound as far as the registers are concerned, but it has not been scheduled. Thus, after this phase, the program contains two types of operations. Firstly, it contains operations that have been narrowed down to fully-qualified operations. Secondly, it contains spill operations whose operands are fully bound to compiler-register tuples, but whose opcodes are still at the level of access-equivalent opcode sets. We call such operations *register-qualified operation sets*.

Post-pass scheduling. A second pass of scheduling, called post-pass scheduling, is necessary to schedule the spill code introduced by the register allocator. This phase has a choice for the scheduling of fully-qualified operations: it can either keep the opcode

[7] In the case of statically scheduled EPIC machines, the binding by the scheduler, of an operation to an opcode-qualified operation set, fully specifies the latency parameters and resource usage. This is a necessary property since the register allocator has to be able to choose any register in the set without affecting the correctness of the schedule created by the scheduler. Thus, all choices have to be identical with respect to their latency parameters and resource usage. It is also important to note that this doesn't apply to dynamically scheduled machines or machines with dynamic arbitration for resources in which, regardless of what the scheduler decides, the hardware may cause an operation to use any one of many resources based on the resource availability at run-time, each one, possibly, with a different latency. However, this is not a problem since the hardware is responsible for preserving the semantic correctness of the program.

bindings selected by the earlier scheduling phase or it can start afresh by reverting all compiler-opcodes back to their original access-equivalent opcode sets. We prefer the latter strategy, since it gives more freedom to the scheduler in accommodating the spill code and yields better schedules. Post-pass scheduling deals with code containing virtual registers that are fully bound to compiler-registers. It is greatly constrained, therefore, by a host of anti- and output dependences. However, since the register assignments were made subsequent to the main scheduling phase, they are already sensitive to achieving a good schedule.

Code emission. The final phase is the code-emission phase. This phase converts fully-qualified operations to architectural operations. This is a book-keeping step and no decisions are made by this phase.

4 Machine description database contents

Based on the model of EPIC code generation and the phase ordering laid out in the previous section, we now discuss the type of information that must be provided in a machine-description database for use by the scheduling and register allocation phases. The following description of the mdes database is primarily for conceptual purposes. Section 5 describes the details of our implementation.

4.1 Operation, opcode, and register descriptors

The EPIC code generator is structured around the process of binding a semantic operation to an architectural operation, each consisting of an opcode and some number of registers operands. The opcode and the registers may each be specified at various levels of binding, from the minimally bound generic opcode or register sets, through the more tightly bound access-equivalent opcode and register sets, to the completely bound compiler-opcodes or compiler-registers.

Corresponding to each opcode set at each level in the OBL, the mdes contains an *opcode descriptor* that specifies the compiler-opcodes contained in that set. It also specifies an opcode attribute descriptor (see Section 4.2.1) which records various semantic attributes of the opcode. In the case of a compiler-opcode, the opcode descriptor also specifies a bundle-macro descriptor (see Section 4.2.2) which, in turn, specifies the implementation of the compiler-opcode using architectural opcodes.

Likewise, for each register set at each level in the OBL, the mdes contains a *register descriptor* that specifies the compiler-registers contained in that set as well as a storage attribute descriptor (see Section 4.3.1) that specifies their storage attributes. For a compiler-register, the register descriptor also specifies a register-package descriptor (see Section 4.3.2) which, in turn, specifies the set of architectural registers that comprise a compiler-register.

As shown in Figure 4, operations are specified at various levels of binding. Only certain combinations of opcode sets and RS-tuples yield legal operation sets. Each legal combination is defined by an *operation descriptor* which specifies an opcode descriptor and as many register descriptors as there are register sets in the RS-tuple. Additionally, for opcode-qualified operation sets and fully-qualified operations, the operation descriptor also specifies a latency descriptor and a reservation table (see Section 4.4).

This constitutes the key information that we wish to emphasize in this report. Although there may be other information that needs to be attached to an operation, opcode or register set, it is not relevant to the topics that we wish to discuss here. We now describe the various items of information that are associated with opcode, register or operation descriptors.

4.2 Information associated with opcodes and opcode sets

4.2.1 Opcode attribute descriptors

Every generic opcode set, access-equivalent opcode set, compiler-opcode and architectural opcode has an associated opcode attribute descriptor which specifies various properties of the opcode or the opcode set which are used during compilation. These include the following:

- The number of source and destination operands.
- If the target machine supports speculative execution, then whether or not the opcode (set) has a speculative version.
- If the target machine supports predicated execution, then whether or not the opcode (set) has a guarding predicate input.
- Opcode classification into certain abstract classes (such as integer, floating-point, memory, branch) for compiler use.

- Certain semantic properties useful during compilation; examples include whether or not the opcode represents an associative operation or a commutative operation.

4.2.2 Bundle-macro descriptors

A compiler-opcode is implemented by one or more architectural opcodes as specified by a bundle-macro descriptor. A *bundle-macro* represents a small computation graph which contains no control flow. The operators in this graph have a fixed issue time relative to one another. This computation graph may contain internal variables that may only be accessed by the operators of the bundle macro. Thus, their lifetime is entirely contained within the duration of execution of the bundle-macro and has a pre-determined length. Internal variables either represent storage resources that are not subject to register allocation or represent architecturally-invisible, transitory storage resources. Some operators may have one or more source operands that are bound to a literal. Source operands that are bound to neither an internal variable nor a literal constitute the source operands of the compiler-opcode of which this bundle-macro is the implementation. Those destination operands that are not bound to an internal variable constitute the destination operands of the compiler-opcode. A bundle-macro specifies the following information:

- A list of architectural opcodes.
- The issue time of each opcode, relative to the issue time of the earliest one.
- A list of internal variables.
- The binding of each operator's source operands to either an internal variable, a literal or a formal input argument of the bundle-macro.
- The binding of each operator's destination operands to an internal variable, a write-only register or a formal output argument of the bundle-macro.

In the simplest and most frequent case, a bundle-macro consists of a single architectural opcode, none of whose operands have been bound to either a literal or an internal variable. In some other cases, it consists of a single architectural opcode, one of whose source operands has been bound to a literal.

4.3 Information associated with registers and register sets

4.3.1 Storage attribute descriptors

Every generic register set, access-equivalent register set, compiler-register and architectural register has an associated storage attribute descriptor which specifies the following five attributes:

- The bit width of the register. This is the number of bits in the container.
- The presence or absence of a speculative tag bit. Architectures that support compile-time speculative code motion (control, data or both) provide an additional speculative tag bit in each register that is intended to participate in the speculative execution. This is provided for the correct handling of architecturally-visible exceptions [13, 14, 8].
- Whether this is part of a static or rotating architectural register file. This determines whether the register specifier used for this register is an absolute register address or an offset from the rotating register base [8].
- Whether the register is read/write (the normal case), read-only (a literal) or write-only (the proverbial "bit bucket", which is often implemented as GPR 0 in a number of architectures).
- For a read-only register, the value of the literal.

Note that the presence or absence of the speculative tag bit and the static/rotating classification make sense only for "normal" architectural registers and not for literals or the bit bucket. By definition, every compiler-register in a generic or access-equivalent register set possesses the same set of storage attributes.

4.3.2 Register-package descriptors

A compiler-register consists of a *register package*, i.e., one or more architectural registers, as specified by the register-package descriptor. The register-package descriptor specifies the following information:

- A list of architectural registers.
- The mapping from the bits of each architectural register to the bits of the compiler-register.

It is important to note that an architectural register can be part of more than one register-package (and hence compiler-register). For example, in the case where an even-odd pair of single-precision floating-point registers is treated like a double-precision compiler-register, the single-precision architectural registers in the pair are both single-precision compiler-registers and elements of a double-precision compiler-register. Most often, however, a compiler-register consists of a single architectural register. This includes the case of a compiler-register such as the stack pointer which might be implemented using (for instance) register 31 in the integer register file.

All architectural registers that are part of a compiler-register must be identical with respect to all of their storage attributes and accessibility from opcodes. The width of a compiler-register is a function of the width of constituent architectural registers. The other attributes simply carry over.

4.4 Information associated with operations and operation sets

4.4.1 Latency descriptors

Each opcode-qualified operation set or fully-qualified operation has, associated with it, a set of latencies that are collectively provided in a *latency descriptor*. All latencies are expressed as the length of the time interval, measured in units of cycles, from the time of initiation of the operation, and are integer valued.

The following two latency values are provided for each register source operand:

- The *earliest read latency* (T_{er}) which is the earliest time at which a particular register source operand could possibly be read, relative to the initiation time of the operation.
- The *latest read latency* (T_{lr}) which is the latest time at which a particular register source operand could possibly be read, relative to the initiation time of the operation.

The following two latency values are provided for each register destination operand:

- The *earliest write latency* (T_{ew}) which is the earliest time at which a particular register destination operand could possibly be written, relative to the initiation time of the operation.
- The *latest write latency* (T_{lw}) which is the latest time at which a particular register destination operand could possibly be written, relative to the initiation time of the operation.

The following latency value is provided for each operation:

- The *operation latency* (T_{op}) which is the time, relative to the initiation time of the operation, at which it completes, i.e., the earliest point in time, ignoring interruptions, after which it will not cause any change of state or make use of any machine resources that are visible to the compiler.

The following two latency values are provided for each load or store operation:

- The *earliest memory serialization latency* (T_{em}) which is the earliest possible time, relative to the initiation time of the memory operation, at which it could reach the point in the memory pipeline beyond which all operations are guaranteed to be processed in FIFO order.
- The *latest memory serialization latency* (T_{lm}) which is the latest possible time, relative to the initiation time of the memory operation, at which it could reach the point in the memory pipeline beyond which all operations are guaranteed to be processed in FIFO order.

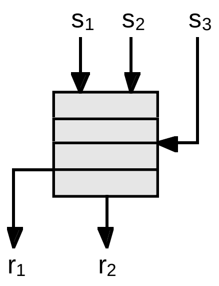
The following latency value is provided globally for all branch operations and is, therefore, not part of any latency descriptor:

- The *branch latency* (T_{br}) which is the time, relative to the initiation time of a branch operation, at which the target of the branch is initiated.

These latencies are discussed at length below.

Since the mdes is the compiler's view of the processor, the latencies for an operation are specified in the abstract context of the scheduler's virtual time. In the scheduler's virtual time, one instruction (possibly consisting of multiple operations) is started per cycle. An abstract clock separates pairs of instructions that are consecutive in time. In this logical world view, one instruction is initiated at the instant of each abstract clock and results are written into destination registers at the instant of an abstract clock, i.e., processor state changes at these instants. Source registers are viewed as being read "just after" the abstract clock specified by the read latency (relative to the initiation time of the operation). This is only a logical view which must be extracted from an examination of the timing relationships of the processor.

Thus if an operation that writes a register and another one that reads that same register are scheduled such that the scheduled time of the read event is the same as (or later than) the scheduled time of the write event, then the result computed by the first one will serve as the operand for the second. If this is not desired, the operations should be scheduled such that the read event is scheduled to occur at least one cycle before the write event. Generally, multiple write events that attempt to change the same item of processor state at the same time, result in an undefined state. Accordingly, such write events must be separated in time by some minimum amount which is usually, but not necessarily, one cycle. In this report we concern ourselves only with processors for which this minimum requisite interval is one cycle. Likewise, we restrict ourselves to processors in which a register read can be initiated every cycle. Note that these restrictions are imposed on our current compilation model and do not affect the mdes infrastructure in any way.



$$\texttt{multadd}(r_1,r_2,s_1,s_2,s_3) \equiv r_1 \leftarrow s_1 \times s_2,\ r_2 \leftarrow r_1 + s_3$$

Figure 5: A temporal representation of a pipelined 3-input, 2-output floating-point multiply-add operation.

Figure 5 shows the example of a pipelined 3-input, 2-output floating-point multiply-add operation. The first two source operands are read sometime during the first cycle after the initiation of the operation. So, their read latency is 0. The third source operand, the one required by the add, is read sometime between two and three cycles after the operation is initiated and, therefore, its read latency is 2. The result of the multiply is written at the end of 3 cycles, while that of the add is written at the end of 4 cycles.

Interruptions, or more specifically interruption handlers that re-execute an operation, complicate the picture. Examples are the page fault handler that re-executes the load or store after processing the page fault, or a floating-point exception handler that attempts to repair the exception. In such cases, the source operand registers must be retained unmodified, until the latest time at which such an exception can be flagged, so that they can be read again, effectively with a latency equivalent to the exception reporting latency. No other operation should be scheduled in such a way that it can end up modifying these source registers prior to this point in time. After that, the source operand registers can be deallocated and reused (at least as far as this operation is concerned). Likewise, an exception handler may also cause a result to be written either early or late relative to the nominal write latency. It is also possible that there is some non-determinacy even in the nominal latencies. For instance, a heavily pipelined floating-point add may normally take two cycles to write its result, but might take an extra cycle to do so if the result needs to be normalized.

This leads to a situation in which, for every input or output operand, one must specify an earliest and a latest read or write latency, respectively. In all cases, when specifying the above latencies, it is the responsibility of the creator of the machine description to account for all the diverse factors that can affect the latencies with which the inputs may be read and the outputs written.

Table 3: Part of the latency descriptor for the 5-operand multiply-add operation in Figure 5.

	Source read latency		**Destination write latency**	
	Earliest (T_{er})	**Latest (T_{lr})**	**Earliest (T_{ew})**	**Latest (T_{lw})**
s_1	0	2	-	-
s_2	0	2	-	-
s_3	2	2	-	-
r_1	-	-	2	3
r_2	-	-	2	4

Once again, consider the multiply-add example of Figure 5. Let us assume that either the multiply or the add may cause an exception at a latency of 2 cycles. Consequently, the source operands may be read by the exception recovery code with an effective latency of 2 cycles. Thus, the latest read latencies for all three source operands is 2 cycles. Assuming that the exception handler computes and writes both results to their respective destinations before returning control, the effective latency with which the destination operands may be written by the exception recovery code is 2 cycles. Consequently, the earliest write latencies for both destination operands must be set to 2 cycles. Table 3 shows part of the latency descriptor for this operation.

The latencies involving the register operands of loads, stores and branches are defined as above. Because of cache faults, memory bank conflicts, etc., the behavior of loads and stores is unpredictable. Therefore, the expected latency with which the result will be written to the destination register, is used as the write latency of a load operation. To reduce the variance between the expected and the actual load latencies, HPL-PD provides different load opcodes that correspond to the level in the cache hierarchy at which the requested datum is expected to be found. The latency of a read from a particular level is used as the latency for the load operations corresponding to that level.

Also, because of the aforementioned unpredictability of memory operations, no attempt is made to model precisely the latency with which the memory location is accessed by a load or store. Instead, the assumption is made that as long as memory operations to the same memory location get to a particular point in the memory system's pipeline in a particular order, they will be processed in that same order. So, in terms of preserving the desired order of memory accesses, the latency that we are concerned with is that of getting to the serialization point in the memory pipeline. We define the latency to the serialization point to be the *memory serialization latency*. In the presence of page faults, one must allow for the possibility that the operation will be re-executed after the page fault has been handled, which leads to some ambiguity in the effective memory serialization latency. To address this, two latencies are associated with each memory operation: the earliest memory serialization latency, T_{em}, and the latest memory serialization latency, T_{lm}.

For fully-qualified operations, the above latency values are derived by directly examining their mapping to architectural opcodes. When a compiler-opcode is implemented by a non-trivial bundle-macro, the computation of these latencies is a bit more intricate, but still unambiguous. (A point to bear in mind here is that the relative schedule times of all the architectural opcodes in the bundle-macro are known.)

For generic and access-equivalent operation sets, the latency values are derived by determining, for each latency, either the smallest or the largest corresponding latency over all fully-qualified operations for that operation set. All four latencies are calculated so that they are optimistic in the sense that they minimize inter-operation delays (as defined in Table 4 later) and, hence, the schedule length. Consequently, T_{lw} and T_{lr} are computed as the *min* of the corresponding latencies over all fully-qualified operations, whereas T_{er} and

T_{ew} are computed as the *max* of the corresponding latencies. Note that the set of latencies computed in this manner may not be achievable, i.e., there may be no single fully-qualified operation that possesses this set of latencies. During scheduling, as each operation is scheduled, and the latencies for that operation are known unambiguously, the delays on each of the edges into and out of that operation need to be corrected[8].

4.4.2 Resources and reservation tables

Each target machine is characterized in the mdes by a set of *machine resources*. A machine resource is any aspect of the target architecture for which over-subscription is possible if not explicitly managed by the compiler. Such over-subscription may either lead to hardware conflicts and incorrect, undefined results, or at the very least result in inefficient execution due to numerous pipeline stalls caused by the arbitration logic. Therefore, it is necessary to model the resource requirements of each opcode-qualified operation set or fully-qualified operation and present it to the compiler in a compact form. The scheduler uses this information to pick conflict-free alternatives for each operation of the computation graph, making sure that no resource is used simultaneously by more than one operation. The three kinds of resources, which are modeled in the mdes, are described below.

Hardware resources are hardware entities that would be occupied or used during the execution of architectural opcodes identified by an opcode-qualified operation set or a fully-qualified operation. This includes integer and floating point ALUs, pipeline stages, register file ports, input and result buses, etc. Note that only those aspects of the target architecture that have the possibility of sharing and resource conflict need to be modeled as independent resources. For instance, in the case of a non-blocking, pipelined ALU, all the pipeline stages need not be modeled separately, modeling the first stage subsumes the possible resource conflicts at other stages. Similarly, only those register file ports and buses need to be modeled that are shared by two or more functional units.

Abstract resources are conceptual entities that are used to model operation conflicts or sharing constraints that do not directly correspond to any hardware resource. This includes combinations of operations that are not allowed to be issued together due to instruction format conflicts such as sharing of an instruction field. In such cases, we create an abstract

[8] At the time of writing this document, Elcor does not do this. Therefore, to preserve correctness, these latencies are actually computed conservatively, i.e., T_{lw} and T_{lr} are computed as the *max* of the corresponding latencies over all fully-qualified operations, whereas T_{er} and T_{ew} are computed as the *min* of the corresponding latencies (see Section 5.1.1).

resource that is used at the same time (e.g. issue time) by the various operations that have a conflict as shown in Figure 6. This prevents the scheduler from scheduling such operations simultaneously.

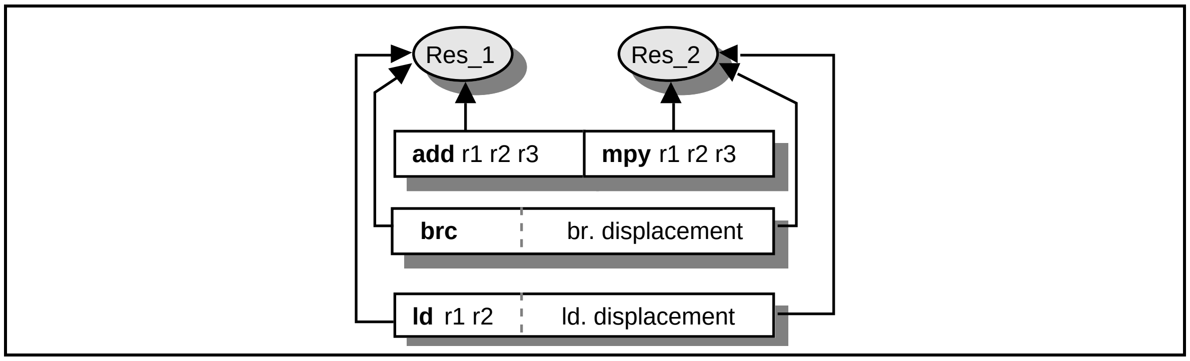


Figure 6: Instruction format conflicts may be modeled using abstract resources. The instruction format may dictate that either a branch with displacement may be issued, or a load with displacement may be issued or an add and a multiply may be issued together. This can be modeled by two abstract resources **Res_1** and **Res_2** as shown. The branch and the load operations each use both resources while the add only uses **Res_1** and the multiply only uses **Res_2**.

Sometimes several identical resources are present in the architecture that may be used interchangeably by an operation. These could be either hardware resources, e.g., any two input buses or any result bus that connects the given functional unit to the appropriate register file ports, or abstract resources, e.g., any issue slot of an instruction for issuing the given operation. It is useful to represent such resources as *counted resources*, i.e., the scheduler is merely required to obey the constraint that the total number of such resources used in a given cycle does not exceed the number available, but the specific resource assigned to an operation is immaterial or may even be determined dynamically.

In addition to defining the set of all machine resources for the purpose of the compiler, the mdes also records how each opcode-qualified operation set or fully-qualified operation uses these resources during specific cycles relative to its initiation time. Collectively, such a table of resource usages is termed a *reservation table* [15]. As an example, we show the pictorial representation of the reservation tables for the Add, Multiply, and Load operations for a hypothetical machine in Figures 7a, 7b and 7c respectively. The tables use hardware resources (ALU, MULT, ResultBus) to model resource conflicts in the datapath and abstract resources (Res_1, Res_2) to model instruction format constraints. Had they been present, counted resources would be marked with the number of resources needed by that operation at that cycle rather than a simple checkmark.

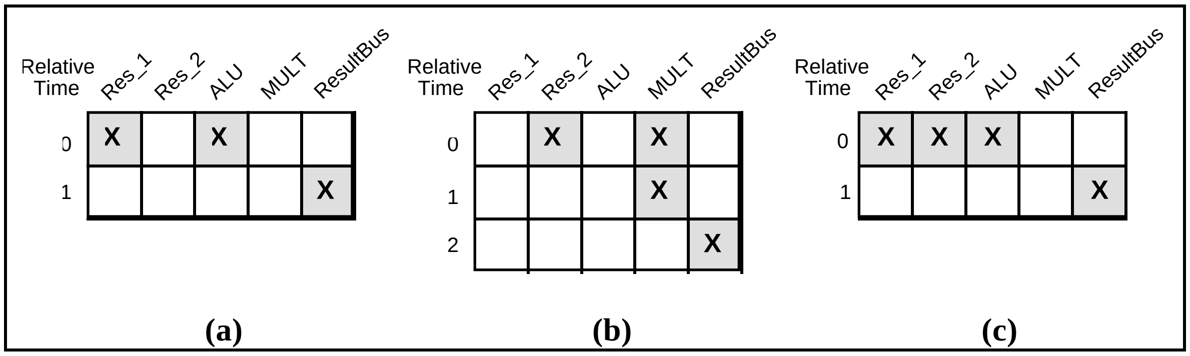


Figure 7: Reservation tables. (a) For an Add operation, which uses Res_1 and ALU at cycle 0 and ResultBus at cycle 1. (b) For a Multiply (Mpy) operation, which is only partially pipelined as it uses MULT resource for two consecutive cycles. (c) For a load operation, which uses the ALU to do address calculation.

Reservation tables can be an economical way of representing the pairwise constraints between alternatives. If N alternatives all make one use each of a resource in their respective reservation tables, N(N+1)/2 pairwise constraints have been specified, implicitly. For instance, by virtue of the fact that all three reservation tables in Figure 7 make use of the ResultBus resource, we have implicitly expressed the three inter-operation constraints that an Add and a Load may not be issued simultaneously, and that neither one may be issued one cycle after a Multiply as well as the three constraints that only one each of the three operations can be issued in any given cycle. If desired, these constraints can be made explicit by computing the *forbidden initiation intervals* between each pair of reservation tables [15].

In a machine with predicated or conditional execution [6, 8], an operation may use certain resources only conditionally (if the operation's predicate operand, which serves to guard the operation, is `true`). Such resource usages may be marked in the reservation table as being conditional so that the scheduler can make the same resource available to another operation under a mutually exclusive predicate.

When the compiler-opcode corresponding to the opcode-qualified operation set or fully-qualified operation is implemented with a non-trivial bundle-macro, the reservation table for the operation is computed by taking the union of the resource usages of the component architectural operations. In principle, reservation tables, too, could be conservatively approximated for access-equivalent and generic operation sets, although the authors are not yet aware of any useful way of doing so.

4.5 Usage of the mdes by the scheduler and register allocator

We now look at how the scheduler and register allocator make use of the above information in the process of translating semantic operations to architectural operations. Our focus here is not on the specific algorithms and heuristics used but, rather, on what each phase is supposed to accomplish, with a view to identifying what information each one needs from the mdes.

4.5.1 Scheduling

The scheduler is the module which uses the most detailed information about the target architecture, its machine resources, its opcode-qualified operation sets, their reservation tables and latencies. The primary task of the scheduler is to select, for each operator, a compiler-opcode from that operator's access-equivalent opcode set and to assign a schedule time to it, subject to the constraints of data dependence and resource availability, while minimizing the schedule length by making full use of the machine resources.

Dependence graph construction. The actual scheduling step is preceded by a preparation step that builds the *data dependence graph* of the region to be scheduled. The vertices of this graph are operations which have been annotated with access-equivalent operation sets, connected by dependence edges representing constraints on scheduling. There are three kinds of dependence edges that result from a pair of operations accessing a register in common: *flow dependence*, *anti-dependence*, and *output dependence* edges. Flow analysis of the region to be scheduled determines the kind of edges that need to be inserted between operations that either define or use the same or overlapping (aliased) registers, either virtual or physical.

Often, the scheduler requires the insertion of a different kind of edge between two operations to specify an ordering between the two that is unrelated to the register operands that are read or written in common by those two operations. These are called sync edges or sync arcs. Sync arcs between a pair of memory operations (loads or stores) specify the ordering that must be maintained between those operations in order to honor the flow, anti- and output dependences that exist between them by virtue of their (possibly) accessing the same memory location. Such sync arcs are termed *Mem edges*. In comparison to the dependences that exist between operations because of their use of a common register, where the presence or absence of the dependence is unambiguous, memory dependences can be ambiguous since the addresses of the referenced memory locations are not always known at compile-time. So, instead of using the regular flow, anti- or output dependences, Mem edges are used.

Sync arcs also exist between a branch and the operations before and after it. These are termed *Control1 edges*. Control1 edges from a branch to the non-speculative operations after it (including other branches) represent control dependences which ensure that those operations will not be issued prior to the branch taking. This is because they depend upon the branch going a particular way and should not, therefore, be issued if the branch goes the other way. The Control1 edges to a branch from the operations before it ensure that those operations are issued, perhaps even completed, before the branch takes.

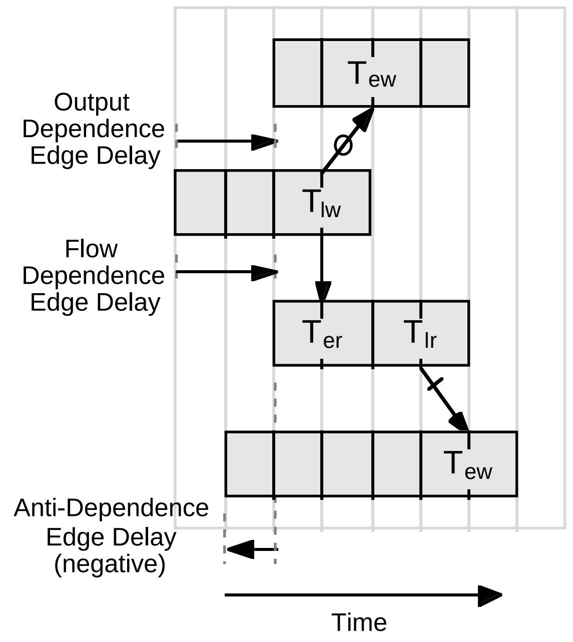
Thus far, we have ignored the possibility that the code may be predicated. If two operations which access the same register are predicated, the possibility arises that their respective predicates are mutually exclusive, i.e., they cannot both be true simultaneously. In such cases, no dependence edge need be drawn between the two operations. Since they cannot both execute during any given traversal of that region, there cannot possibly be any interference caused as a result of both accessing the same register. However, it is still correct, albeit conservative, if edges are inserted as if the code was unpredicated.

Edge delay computation. Each dependence edge is decorated with an *edge delay* that specifies the minimum number of cycles necessary, between the initiation of the predecessor operation and the initiation of the successor operation, in order to satisfy the dependence. The manner in which the edge delays are computed depends on whether the operations function under the freeze or drain interruption model.

Consider first the formulae for computing edge delays when the successor operation uses the freeze model, regardless of which model the predecessor uses (Table 4). These formulae are best understood by referring to Figure 8 and the dependence constraints below, where t(X) is the time at which operation X is initiated, P is the predecessor operation and S is the successor operation.

Flow dependence:	$t(S) + T_{er}(S) \geq t(P) + T_{lw}(P)$
Anti-dependence:	$t(S) + T_{ew}(S) \geq t(P) + T_{lr}(P) + 1$
Output dependence:	$t(S) + T_{ew}(S) \geq t(P) + T_{lw}(P) + 1$

For each type of dependence edge, the delay is the minimum value of t(S) - t(P) that is permitted by the corresponding constraint.



T_{ew}
Output Dependence Edge Delay
T_{lw}
Flow Dependence Edge Delay
T_{er}
T_{lr}
T_{ew}
Anti-Dependence Edge Delay (negative)
Time

Figure 8: Edge delays for the three types of data dependences.

In light of the definition of these latencies, the operation producing a result writes the datum to its destination register no later than T_{lw} cycles from the initiation of that operation, whereas the operation consuming that datum reads its source register no earlier than T_{er} cycles from the initiation of that operation. Consequently, in the case of a flow dependence, the earliest time at which the successor operation reads its input may be scheduled to be as early as, but no earlier than, the latest time at which the predecessor operation writes its result. The flow dependence edge delay is, therefore, the difference between the latest write latency and the earliest read latency of the predecessor and successor operations, respectively. Likewise, as explained in Section 4.4, an anti-dependence (output dependence) means that the earliest time at which the successor operation writes its result may be scheduled to be no earlier than one cycle after the latest time at which the predecessor operation reads its input (writes its result).

From Table 4 and Figure 8, we see that edge delays may turn out to be positive or negative depending on the latencies of the operands. A negative edge delay provides greater scheduling freedom for the successor operation, i.e. the successor operation may even be initiated earlier than the predecessor operation and still meet the data dependence constraint.

Table 4: Computation of edge delays for the various types of dependences. In this table, P represents the predecessor operation and S represents the successor operation. The scheduler must satisfy the constraint that t(S) - t(P) is greater than or equal to the delay as computed according to this table, where t(X) represents the time at which operation X is scheduled to start execution.

Type of dependence	**Edge Delays when S operates with the Freeze Model**	**Edge Delays when S operates with the Drain Model**
Flow dependence	$T_{lw}(P) - T_{er}(S)$	$\max[0, T_{lw}(P) - T_{er}(S)]$
Anti-dependence	$T_{lr}(P) - T_{ew}(S) + 1$	$\max[0, T_{lr}(P) - T_{ew}(S) + 1]$
Output dependence	$T_{lw}(P) - T_{ew}(S) + 1$	$\max[0, T_{lw}(P) - T_{ew}(S) + 1]$

Edge delays are computed slightly differently for machines that drain their pipelines on fielding an interruption. Draining the pipelines causes the operations that are currently in flight to read their inputs and write their results, in effect, at the time that the interruption takes place and, possibly, *before* their scheduled latency in terms of the schedule's virtual time. Predecessor operations that were scheduled to be initiated after the point of interruption will not be issued until after the interruption has been serviced and well after their successor operations have been drained from their pipelines. Consequently, they can end up performing their reads and writes *after* the reads and writes that they were supposed to precede, thereby violating the semantics of the program.

This problem is avoided by the enforcement of a simple rule--that a successor operation never be scheduled to be initiated earlier than a predecessor operation, if that successor operation operates under the drain model. If the successor operation operates with the freeze model, the rule permits the predecessor to be scheduled later than the successor. If this rule is enforced, it is impossible for an interruption to cause the successor operation to be issued and perform all of its processor state changes without the predecessor operation having been issued, either concurrently or earlier. Consequently, the interruption cannot result in any dependence constraint being violated. With this in mind, the edge delays are adjusted with the following additional constraint:

$$t(S) \geq t(P).$$

For each type of dependence edge, the only difference from the freeze case is that the edge delay is permitted to be no less than 0 (as specified in the rightmost column of Table 4).

The dependence edges due to the register accesses of loads, stores and branches have their edge delays computed as described above, and the latencies needed to compute them are provided in their latency descriptors just as for other operations. However, sync arcs are treated differently. The assumption is made that as long as memory operations to the same memory location get to a particular point in the memory system's pipeline in a particular order, they will be processed in that same order. So, with a view to preserving the desired order of memory accesses, but ignoring address-related exceptions, the delay on the Mem edge between two memory operations that may be to the same memory location should be set to

$$T_m (P) - T_m (S) + 1,$$

where T_m is the memory serialization latency, and where P and S are the predecessor and successor operations, respectively.

In the presence of page faults, one must allow for the possibility that the predecessor operation will be re-executed after the page fault has been handled, which leads to some ambiguity in the memory serialization latency. To address this, the edge delay should be set to

$$T_{lm} (P) - T_{em} (S) + 1.$$

The above formula applies to the freeze case. In the drain case, it is further constrained to be non-negative.

If the hardware supports prioritized memory operations [8] then these edge delays can be set to 0. This is advisable only if it is statistically likely that the operations are to different memory locations, even though this cannot be proved at compile-time.

Next, consider Control1 edges from a branch to those non-speculative operations that should not be issued until after the branch has completed. Included are all operations that depend upon the branch going a particular way and which should not, therefore, be issued if the branch goes the other way. Also, a subsequent branch operation whose branch condition is not mutually exclusive with that of the preceding branch must be constrained to not issue until the preceding branch has completed[9]. In both cases, the edge delay is set to the branch latency, T_{br}, of the predecessor branch. This edge delay does not depend upon the interruption model used by the successor operations.

Finally, consider the Control1 edges to a branch from the operations that precede it in the sequential ordering. For correct execution, one must guarantee that all of these operations are issued before the branch operation completes. (If not, if the branch takes, then some of these operations which were supposed to have been executed will not even have been issued, and they never will.) This leads to a possibly negative delay between each such operation and the branch as given by $(1 - T_{br})$. Note that since branch operations always adhere to the freeze model, we do not need a second formula for the drain case.

More conservatively, one can require that all preceding operations have completed by the time the branch completes. If so, the edge delay is given by $(T_{op} - T_{br})$, where T_{op} is the operation latency of the predecessor operation.

Tracking resource usage. Given a data dependence graph, properly decorated with edge delays, the actual task of scheduling can begin. The reservation tables of the various opcode-qualified operation sets and fully-qualified operations are used during scheduling to prevent scheduling of alternatives in a conflicting manner. Consider the example computation graph shown in Figure 9a. Each operator in the graph is scheduled in some priority order (based on data dependence) by choosing a compiler-opcode from its assigned access-equivalent opcode set. The choice is made such that the reservation table of the selected alternative (Figure 7), offset by the proposed initiation time of the operation, does not conflict with any of the previously reserved resources. For this purpose, the scheduler internally maintains a *resource usage map (RU map)* which keeps track of when each machine resource will be used by the previously scheduled operations (Figure 9b). In the example shown, suppose the operations are scheduled in the order A1, L, M, A2. The operation L cannot be scheduled at cycle 0 since its reservation table will conflict with the previously scheduled operation A1 at resource positions Res_1 (cycle 0), ALU (cycle 0), and ResultBus (cycle 1). On the other hand, after scheduling A1, L and M at times 0, 1 and 2, respectively, the operation A2 can fit nicely into the same cycle as M since it does not cause a resource conflict. Figure 9c shows the final schedule.

[9] If the branch conditions are known to be mutually exclusive, the edge between the two branches can be deleted [16].

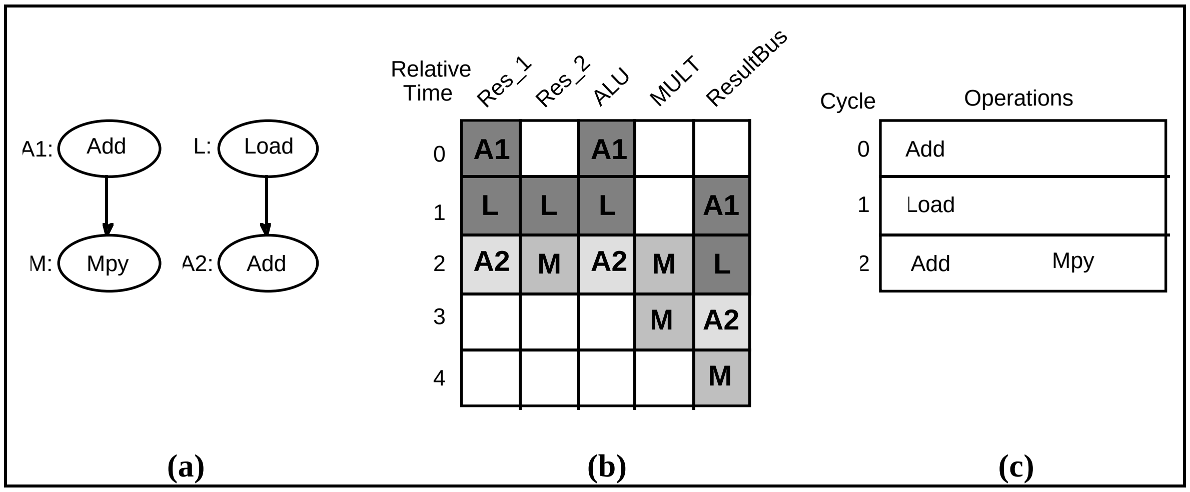


Figure 9: Resource tracking using the resource usage map. (a) A simple computation graph to be scheduled. (b) The resource usage map which keeps track of the partial schedules during scheduling. The figure shows the map after all four operations has been scheduled using the reservation tables of Figure 7. The operation A1 is scheduled to start at time 0, M at time 2, A2 at time 2 and L at time 1. (c) The final schedule for the computation graph shown in (a).

4.5.2 Register allocation and spill code insertion

In our phase ordering, scheduling is performed before register allocation, and the access-equivalent operation sets have been bound to opcode-qualified operation sets. The job of the register allocator is to take virtual registers, that have been partially bound to access-equivalent register sets, and bind them to compiler-registers. In the process, it binds each opcode-qualified operation set to a fully-qualified operation. We shall restrict our discussion to the case in which either the code is not predicated or the interference graph construction step of register allocation is performed in a conservative fashion, essentially acting as if the code was not predicated. For a discussion of register allocation of predicated code, the reader is directed to the literature [17, 18].

Interference graph construction. Two virtual registers may be assigned the same compiler-register if doing so does not alter the flow dependences of any of the operations involved and if for the new anti- or output dependences that are introduced, no edge delays (as described earlier in this section) are violated by the pre-existing schedule. Register allocation is performed with the help of an interference graph which consists of a vertex per virtual register. If two virtual registers may not be assigned the same compiler-register, then an interference edge is inserted between the corresponding vertices in the interference graph. Register allocation is then formulated as a graph coloring problem on this interference graph.

We consider first the case of the freeze model. Let A and B be two virtual registers. Let W_A and W_B represent the set of operations that write to A and B, respectively. Likewise, let R_A and R_B represent the set of operations that read from A and B, respectively. If A and B are to be assigned the same compiler-register without altering any of the flow dependences, all of the reads and writes to one of the virtual registers (the first one) must precede all of the writes to the other virtual register (the second one) by at least one cycle[10]. If not, at least one of the readers of either A or B will get the wrong value. Without loss of generality, assume that A is the first lifetime. This constraint can then be stated as

$$\min[t(W_{Bi})+T_{ew}(W_{Bi})] \geq \max[t(R_{Aq})+T_{lr}(R_{Aq}),\ t(W_{Ap})+T_{lw}(W_{Ap})] + 1$$

across all W_{Bi}, W_{Ap} and R_{Aq} that are members of W_B, W_A and R_A, respectively, and where t(X) is the scheduled initiation time of operation X. If the edge delay on every newly introduced anti- or output dependence edge delay is to be satisfied, so that no interference edge need be placed between A and B, the following two inequalities must be satisfied:

$$\text{Anti-dependence:}\quad t(W_{Bi}) + T_{ew}(W_{Bi}) \geq t(R_{Aq}) + T_{lr}(R_{Aq}) + 1$$
$$\text{Output dependence:}\quad t(W_{Bi}) + T_{ew}(W_{Bi}) \geq t(W_{Ap}) + T_{lw}(W_{Ap}) + 1$$

for any W_{Bi}, W_{Ap} and R_{Aq} that are members of W_B, W_A and R_A, respectively.

Since the two inequalities must be satisfied between every member of W_B and every member of either R_A or W_A, respectively, the two conditions that must both be true for there to be no interference edge between A and B are:

$$\text{Anti-dependences:}\quad \min[t(W_{Bi})+T_{ew}(W_{Bi})] \geq \max[t(R_{Aq})+T_{lr}(R_{Aq})] + 1$$
$$\text{Output dependences:}\quad \min[t(W_{Bi})+T_{ew}(W_{Bi})] \geq \max[t(W_{Ap})+T_{lw}(W_{Ap})] + 1$$

which can be combined into the single inequality:

$$\min[t(W_{Bi})+T_{ew}(W_{Bi})] \geq \max[t(R_{Aq})+T_{lr}(R_{Aq}),\ t(W_{Ap})+T_{lw}(W_{Ap})] + 1$$

over all i, p and q. Note that this is the same constraint as the one needed to ensure that no flow dependence is altered.

[10] Note that in this case no flow dependence can be introduced by assigning the two virtual registers to the same compiler-register as long as the second virtual register is written prior to it being first read. If this assumption is false, then the offending operation is reading an uninitialized virtual register, and we take the position that it does not matter which value this read yields. In other words, although the flow dependence for this offending operation has changed, we take the position that the program semantics have not been altered in any material way.

Table 5: Computation of register lifetimes and initiation lifetimes for a virtual register. t(X) is the time at which operation X is scheduled to start execution. i ranges over all operations that write to that virtual register and j ranges over all operations that read from that virtual register.

Register Birth Time	min[$t(W_i)+T_{ew}(W_i)$],	over all i
Register Death Time	max[$t(W_i)+T_{lw}(W_i)$, $t(R_j)+T_{lr}(R_j)$] + 1,	over all i, j
Initiation Birth Time	min[$t(W_i)$],	over all i such that W_i drains
Initiation Death Time	max[$t(W_i)$, $t(R_j)$],	over all i, j

Instead of computing the terms on the two sides of the inequality repeatedly for each pair of virtual registers, one may observe that the left hand side is a function only of B, and the right hand side is a function only of A. These two terms can be computed, once and for all, for each virtual register as shown in Table 5. The *register birth time* is the earliest time at which any of the operations that write to that virtual register can end up doing so. The earliest time at which an operation writes a result is computed as the sum of the operation's scheduled initiation time and the earliest write latency, T_{ew}, for that destination operand. The *register death time* is one cycle more than the latest time at which any of the operations that read from or write to that virtual register can do so. The latest time at which an operation accesses a virtual register is computed as the sum of that operation's scheduled initiation time and either the latest read latency, T_{lr}, if it is an input operand or the latest write latency, T_{lw}, if it is a destination operand. In order to ascertain these latencies, the latency descriptors associated with the opcode-qualified operation sets are consulted. The time interval between the birth and death times is the *register lifetime* of that virtual register (see Figure 10).

Using the concept of register lifetimes, the interference criterion may be expressed as follows:

> two virtual registers have an interference edge between them in the interference graph if and only if their register lifetimes overlap.

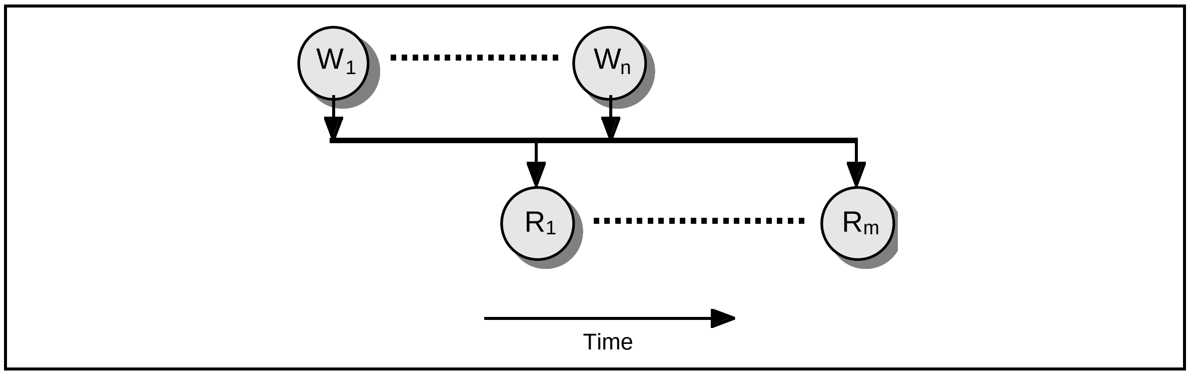


Figure 10: The register lifetime of a virtual register as determined by the times at which it might be written by producer operations $W_1,..,W_n$, and the times at which it might be read by consumer operations $R_1,..,R_m$.

This apparently convoluted way of deriving something very similar to the well-known condition for interference provides us the mechanism for dealing with the drain model as well. Let Y_A represent the set of operations that either read or write A. Let X_B represents the set of operations that both write B and drain on the occurrence of an interruption. Referring to Table 4, we see that for the drain model an additional condition must be satisfied if an interference edge is not to be inserted between virtual registers A and B. Once again assuming that the earliest scheduled operation that accesses A is scheduled earlier than the earliest scheduled operation that writes to B, we have the following additional inequality that must be satisfied if an interference edge is to not be inserted:

$$t(X_{Bi}) \geq t(Y_{Ap})$$

for any X_{Bi} and Y_{Ap} that are members of X_B and Y_A, respectively. Since the inequality must be satisfied between every such member of X_B and every member of Y_A, the additional condition that must be true for there to be no interference edge between A and B is:

$$\min[t(X_{Bi})] \geq \max[t(Y_{Ap})]$$

over all i and p[11].

As before, we define the *initiation birth time* as the earliest initiation time of any of the operations that both write to that virtual register and drain on the occurrence of an interruption (Table 5). The *initiation death time* is the latest initiation time of any of the operations, that either read from or write to that virtual register. The time interval between these two times is the *initiation lifetime* of that virtual register.

[11] This constraint should be understood to be non-existent if the set X_B is empty.

Then, in the case that at least one of the operations that write the second (later) lifetime drain on the occurrence of an interruption, the interference criterion may be expressed as follows:

> two virtual registers have an interference edge between them in the interference graph if and only if either their register lifetimes overlap or their initiation lifetimes overlap.

The register allocation phase is preceded by a lifetime analysis step in which the register lifetime and, if necessary, the initiation lifetime, of each virtual register are computed, which enables the construction of the interference graph. After that, register allocation is performed using graph coloring. Each virtual register is constrained to be colored with (i.e., to be assigned) only the compiler-registers in its access-equivalent register set. Virtual registers with an interference edge between them must not be colored with either the same compiler-register or with compiler-registers that have one or more bits in common. To ascertain this, the register-package descriptor is consulted. If the available compiler-registers are not sufficient to color all the virtual registers, then some of the virtual registers are spilled to memory by inserting the appropriate spill code (stores and loads). Again, any number of heuristics exist for determining an efficient allocation which attempts to minimize the amount of spill code.

5 Elcor’s mdes Query System (mQS)

We now present a brief description of the organization of the machine-description query system in Elcor, our ILP research compiler for the HPL-PD family of architectures [8]. The contents and organization of Elcor's machine-description system is influenced by the fact that our current implementation differs slightly from the general theory described thus far. These differences are discussed in Section 5.1. Additionally, the structure of the database is highly customized to support our preferred module phase ordering and their queries (especially scheduler queries) very efficiently. As such, it broadly conforms to the OBL shown in Figure 4, but reflects a number of optimizations that are described in Section 5.2. The rest of this section describes the various internal data structures of the mdes query system.

5.1 Current differences from the general model

5.1.1 Restrictions and limitations

Table 6 lists the current restrictions and limitations of Elcor and indicates which part of the mdes-driven capability is the cause.

The semantic operations and the code selection process, including knowledge of the generic operation sets, opcode sets and register sets, are currently hard-coded within the Elcor compiler. Elcor, as currently implemented, also only considers single-cluster processors or, more precisely, it assumes that there is a single access-equivalent register set per generic register set. As a result, the pre-pass operation binding phase degenerates to a trivial procedure which is hard-coded into Elcor (and which does not consult the mdes). Also, there has been little attention given in Elcor to code emission, assembly code generation and the assembly process including instruction template assignment and encoding. Consequently, information in support of code selection, pre-pass operation binding and code emission does not show up in the mdes. The mQS is currently designed to support only the scheduling and register allocation phases.

Currently, there is no support for bundle-macros and register-packages. The class of machines that are handled are those with a one-to-one mapping from compiler-opcodes to architectural opcodes, from compiler-registers to architectural registers, and from fully-qualified operations to architectural operations. Therefore, the bundle-macro and register-package descriptors are not present in the database.

A current limitation of the mQS interface and the compiler is that they assume that there is just one alternative for a given compiler-opcode and IO descriptor. This is reflected in the use of the compiler-opcode name as the handle for the alternative chosen during scheduling. The Hmdes2 language and the mdes database, on the other hand, are not constrained in this way.

The current implementation of Elcor's latency descriptors differs in many ways from the description in Section 4.4.1. First, there is only a single, common T_{lr} latency for all of the data input ports. Second, T_{br} is not provided globally, instead, the same port that is used to record T_{lm} for memory operations is used to record T_{br} for branch operations. Third, the latency descriptor does not currently provide T_{op}, which is therefore best computed as the largest of the latest write latencies, T_{lw}, over all the register output ports of that operation.

The mQS interface, however, provides a global upper bound on T_{op} over all operations as a separate constant (`MDES_max_op_lat`) for the compiler to use internally.

Table 6: A list of the current limitations of Elcor's mdes-driven capability, relative to the desired capability as described in Section 4. Also indicated is the part of the machine-description system which currently imposes the limitation: the Hmdes2 language (H), the mdes database (M), the mQS interface (I), and Elcor's compiler modules (E).

Restriction	H	M	I	E
No support for mdes-driven code selection and code emission	x	x	x	x
No support for pre-pass operation binding			x	x
No support for bundle-macros and register-packages	x	x	x	x
Only a single alternative for a given compiler-opcode and IO descriptor			x	x
A single, common T_{lr} for all the data input operands of an operation	x	x		
T_{br} not provided globally, instead, it is shared with T_{lm}	x	x		
T_{op} is not provided	x	x	x	x
Result latencies are identical regardless of where the consuming operation is executed	x	x	x	x
Conservative (as opposed to optimistic) latency summary for acc. eq. ops			x	x
No latency summary for generic operations			x	
Inability to deal with aliased (overlapping) compiler-registers		x	x	
No predicated or counted resource usage	x	x	x	x

With regard to latency queries, a current restriction of the mQS interface is that conservative summaries are generated for latencies of access-equivalent operations rather than optimistic summaries. This may lead to sub-optimal schedules. Also, latency summaries for generic operations are not provided.

Another latency-related limitation built into all components of the machine-description system is that there is a single latency for an operation, regardless of on which functional units the producing and consuming operations are scheduled. Thus a processor on which there is one latency if both operations are scheduled on the same functional unit, but a longer one if they are on separate functional units, is not supported.

The mQS is unable to deal with aliased (overlapping) compiler-registers or architectural registers, i.e., two registers that have at least one bit in common. This includes the case of a double-precision register which consists of an even-odd pair single-precision registers, or a 32-bit control register to access 32, 1-bit predicate registers "broad-side" [8]. As a result, the register section in the Hmdes2 machine description, if present, is ignored. Elcor modules, on the other hand, can handle a limited form of register aliasing without using the mdes facility.

Another feature that is not currently supported, by any part of the system, is the ability to model and take advantage of predicated resource usage. This would enable the scheduler, for instance, to share resources between operations on the two sides of a conditional expression within the same cycle. Support for identical counted resources is also not provided which would have allowed, for instance, modeling superscalar architectures within the same mdes framework.

5.1.2 Additional Facilities

The current mdes implementation provides some additional facilities that are orthogonal to the discussion in Section 4. We describe these below.

The mdes capability of Elcor is "region-based", i.e., different regions of the program may potentially be compiled using different mdeses. A situation where this need arises naturally is when an architecture has more than one modes of operation that differ in their programming model, e.g., user mode vs. supervisor mode, full instruction set vs. a subset instruction set, main processor architecture vs. co-processor architecture etc. In these situations, a mixed mode program needs to be compiled with the different modes in mind. The opcode repertoire, the register space, the machine resources and the latencies may all potentially be different in different modes. Therefore it is best to represent them as separate mdeses. An appropriate mdes may be attached to each program region and all phases of the compiler use that mdes for that region.

The discussion in Section 4 does not take any position on which alternative the scheduler is allowed to pick from a given access-equivalent operation set other than the fact that the resources used by that alternative must be available. On the other hand, some hardware architectures assign execution priorities to various functional units and the scheduler needs to be aware of them so that it can pick the right alternative among the ones that are available. The HPL-PD architecture permits the specification of a relative priority among the various memory ports (load/store units) [8]. In a given machine, the memory ports either may or may not be prioritized. This priority is recorded with the compiler-opcodes associated with that memory unit. In the Hmdes2 specification, however, it is possible to specify that all execution units are prioritized but no use is currently made of this capability in Elcor.

For certain analyses, a lower bound of the resource requirements of a given set of operations may be desired (e.g., the computation of the resource-limited bound, ResMII, on the initiation interval for modulo scheduling of loops [19]). Complete scheduling is far too accurate and slow for this purpose. The *resource minimum schedule length map (RMSL map)* provides an alternate interface that may be used to compute a lower bound on the resource usage. An RMSL map is similar to an RU map of Section 4.5.1 except that it only keeps the total number of the various resources needed by the selected schedule-equivalent operation sets rather than laying them out in a 2-dimensional table. This is an extension of the mdes query interface and does not reflect any change in the external mdes specification.

5.2 Machine-description database optimizations

An efficient implementation of a machine-description database differs quite substantially from the most direct and obvious organization that reflects the OBL. The improved organization can be viewed as the result of a series of optimizations with the naive organization as the starting point. We outline here the set of optimizations applied to the structure of the mdes in our implementation of it within Elcor. Except for one optimization which is aimed at speeding up the queries made by the scheduler, their objective is to reduce the size of the database.

Typically, the most time-consuming part of EPIC code generation is scheduling. It makes sense, therefore, to accelerate the queries made by the scheduler. The most frequent query involves inspecting the set of alternatives corresponding to a given access-equivalent operation set. A natural choice for the representation of the OBL would be as a tree structure rooted in each access-equivalent operation set. The children would be the opcode-qualified operation sets, each of whose children would represent the alternatives. Executing this query would entail a relatively slow walk over this tree. Instead, in our implementation, the access-equivalent operation sets point directly to a list of the corresponding alternatives, each of which has a reverse pointer to the opcode-qualified operation set for which it is an alternative. This speeds up the query, but at the cost of significant redundancy. Fortunately, the next optimization partially compensates for this.

The elimination of redundant information is always important. In the database, it results in reduced size. In the Hmdes2 specification, it leads to conciseness and a better ability to maintain consistency and correctness. The solution, which also serves as a space optimization, is to replace all of the duplicate items of information with pointers to a single copy. Examples of information that typically are highly redundant and that benefit greatly from this optimization include the latency descriptors and the reservation tables within the operation descriptors. Often, this optimization creates further opportunities for itself. For instance, two alternatives, A and B, in different access-equivalent operation sets, that are identical except that they point to different, but identical, latency descriptors, C and D, respectively, become redundant once C and D have been replaced by pointers to the same record. A and B themselves can now be combined since they are identical.

Another space saving optimization is Cartesian factoring. One can take advantage of this opportunity when a (large) set of tuples can be represented as a tuple (Cartesian product) of (small) sets. The primary application of this optimization is to RS-tuples. Consider a set of operation sets that are identical except for their RS-tuples. First, the common part of the operation sets can be factored out using the previous optimization. Then the set of RS-tuples for each operation set can be reduced to a tuple whose elements are sets of register sets. The resulting representation for the entire set of operation sets is not much larger than that for a single operation set.

One way to save space is simply to not represent certain information. For instance, fully-qualified operations and register-qualified operation sets are not represented in the mdes. The register tuple for an operation for which register allocation has been performed is maintained in the compiler's intermediate representation (outside the mdes). The contextual knowledge of the register tuple to which an operation is bound, allows each opcode-qualified or access-equivalent operation set to also represent a fully-qualified operation or a register-qualified operation, respectively.

A final optimization is to replace data that is infrequently or never queried, by a function that assembles that data on demand. One good example is the opcode set of an operation set. This is never examined since the scheduler works instead with the list of alternatives for that operation set. The opcode set need not be represented explicitly in the database except at the leaves where the opcode set degenerates to a compiler-opcode. If it is needed, the opcode set can be constructed by walking the tree rooted in the operation set and assembling the compiler-opcodes at the leaves. In a sense, this is the reverse of the first optimization.

5.3 Elcor's machine-description database (mdes)

The internal organization of Elcor's mdes database is shown in Figure 11. The database is organized as a giant hashtable that is keyed on the name (represented as a string) of a generic opcode set or compiler-opcode. The database is structured in such a fashion that the sequence of database record types that one traverses is identical whether the key is a generic opcode set or a compiler-opcode. This avoids the need for writing code with case statements at every step. We describe the various components of the database below.

Opcode descriptor. The database record resulting from the keyed access to the database is the *opcode descriptor* for either a generic opcode set or a compiler-opcode, depending on the name supplied as the argument. In both cases, the opcode descriptor directly contains the various attributes of the corresponding generic opcode set or compiler-opcode as described in Section 4.2.1.

Each opcode descriptor for a generic opcode set (compiler-opcode) points to a set of access-equivalent (opcode-qualifed) operation sets, which is represented as a list of (smaller) subsets of access-equivalent (opcode-qualified) operation sets that share the same set of alternatives and whose access-equivalent RS-tuples can be compactly represented by an IO descriptor. These subsets are called *IO list elements* and the entire list is called an *IO list*. Each IO list element consists of a pointer to an IO descriptor that identifies the subset, a pointer to a list of alternatives present in that subset and a pointer to the next IO list element.

IO descriptor. An *IO descriptor* is a tuple of IO sets. An *IO set* is a set of access-equivalent register sets which need not necessarily be subsets of the same generic register set. The IO descriptor therefore represents the set of access-equivalent RS-tuples that would be obtained by taking the Cartesian product of its IO sets. As such, it is a very compact representation of a set of access-equivalent RS-tuples that are compatible with the same set of alternatives. This set of compatible alternatives is specified by the list of alternatives pointed to by the parent IO list element. Multiple IO descriptors which are compatible with the same set of alternatives can also point to and share the same list of alternatives.

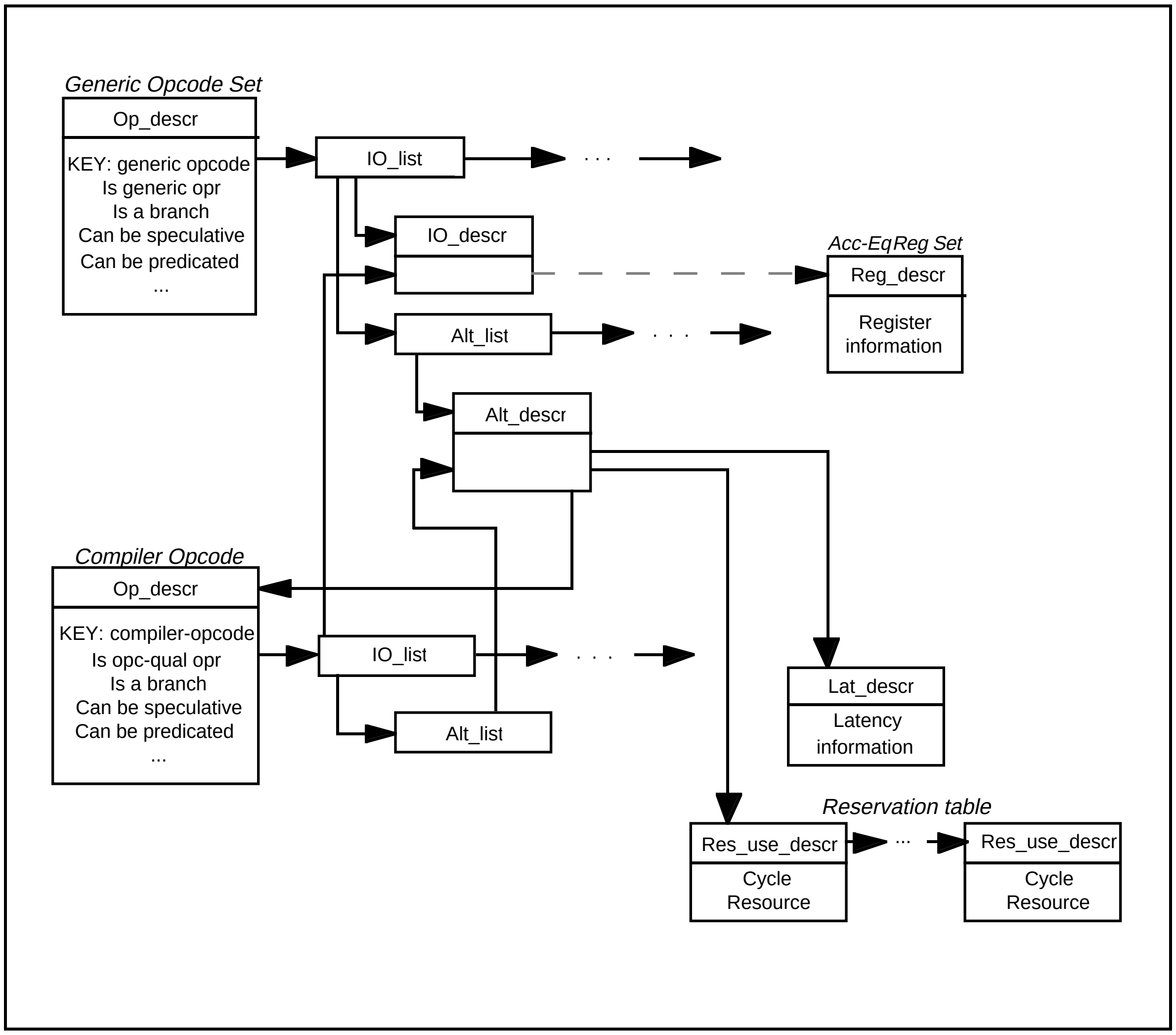


Figure 11: The internal organization of the mdes database in Elcor.

Each access-equivalent RS-tuple in an IO descriptor, in conjunction with the entire set of compatible alternatives associated with it, constitutes an access-equivalent operation set. In this respect, each IO descriptor in an IO list identifies the subset of access-equivalent operation sets that differ only in their access-equivalent RS-tuples and that can together be represented in a factored form. Similarly, each access-equivalent RS-tuple in an IO descriptor, in conjunction with each alternative within its compatible alternative set, constitutes an opcode-qualified operation set.

By the time scheduling starts, the operands of each operation have been bound to access-equivalent register option sets (see Section 3.2.4). This RS-tuple is termed the *IO requirement*. Along with the specification of a generic opcode set (compiler-opcode), it determines the access-equivalent (opcode-qualified) operation set to which the operation in question is bound. The IO requirement is used to select matching IO descriptors from the IO list of the given generic opcode set (compiler-opcode). A match occurs when the RS-tuple denoted by the IO requirement is a member of the set of RS-tuples denoted by the IO descriptor. In general, there may be multiple IO descriptors that match the IO requirements[12]. The IO requirement, along with the list of alternatives corresponding to the matching IO descriptor, constitute the set of access-equivalent (opcode-qualified) operation sets available to the scheduler for an operation with the given generic opcode set (compiler-opcode) and IO requirement.

Alternative descriptor. In addition to the IO descriptor, each IO list element points to a list of alternative list elements, each of which points to an alternative descriptor. An *alternative descriptor* consists of pointers to a reservation table, a latency descriptor and the opcode descriptor for a compiler-opcode. This permits multiple alternatives to share these objects. In turn, an alternative descriptor can be pointed to by elements of multiple alternative lists.

The primary motivation for this organization is to speedup scheduling queries. During scheduling, the compiler is only concerned with the latency descriptor and the reservation table for each available opcode-qualified operation set in order to decide which one to pick. The alternative descriptors provide access to this information as directly as possible. The opcode descriptor for the compiler-opcode that is pointed to by the alternative descriptor provides all the compiler-opcode's attributes. These are needed before and after, but not during, scheduling.

Reservation table. A *reservation table* is represented as a linked list of *resource usage descriptors* each of which specifies a machine resource, the time at which it is used by an operation, and a pointer to the next resource usage descriptor. Since multiple operations

[12] As currently implemented, a query will select the first matching IO descriptor.

that execute on a given functional unit can have the same reservation table, this data structure is shared across multiple operations which may even have different opcodes.

Latency descriptor. The *latency descriptor* specifies all of the operand and operation latencies for a given operation. Since multiple operations, even those that execute on different functional units, can have the same latencies, this data structure is shared across multiple operations.

Register file descriptor. Elcor's mdes does not explicitly represent the entire register set hierarchy of generic and access-equivalent register sets, IO sets and compiler-registers. The only level in the hierarchy that is represented is that of the access-equivalent register set. The *register file descriptor*[13] specifies the number of registers in an access-equivalent register set, and the names (represented as strings) of the registers. Also, the common storage attributes of all these registers as described in Section 4.3.1 are factored and attached to the register file descriptor rather than to each register.

Register descriptor. At the bottom of hierarchy are the compiler-registers whose various attributes ought to be specified in a *register descriptor*. Currently, Elcor's mdes does not represent these descriptors internally. However, they are present in the external mdes specification (Section 7.1). In addition to the storage attributes (which have been factored out and attached to the register file descriptor), the register descriptor would specify register-package information (Section 4.3.2) and whether there is a structural overlap between this compiler-register and any others. Since register-packages and register aliasing is currently not supported in Elcor, this section in the external specification is ignored.

5.4 The resource usage map (RU Map)

As described in Section 4.5.1, the scheduler keeps track of the resource usage of the operations that have already been scheduled using a resource usage map (RU Map). It is a matrix of size (number of cycles in the schedule $\times$ number of resources in the machine). Each entry in the matrix contains a pointer to the operation that has been scheduled to use that resource at that time. Keeping pointers to operations makes it easy to backtrack, i.e., unschedule an already scheduled operation.

[13] Currently, there is a one-to-one mapping between HPL-PD's access-equivalent register sets and its architectural register files. This also explains the choice of name for this descriptor. In Figure 11, however, these are called "Reg_descr".

Rather than directly expose the internal structure of the database or the details of the machine resources and reservation tables to the scheduler, we provide an independent *resource usage manager (RU manager)*, that is part of the mQS, and which does all the book-keeping on behalf of the scheduler. The RU manager is responsible for directly interfacing with the mdes database and allocating and manipulating the resource usage map as instructed by the scheduler; it does not make any scheduling decisions on its own. The RU manager interface queries appear in Section 6.2.3.

5.4 The resource minimum schedule length map (RMSL Map)

As described in Section 5.1.2, the scheduler uses a resource minimum schedule length map (RMSL map) for a quick assessment of the resource requirements of a program fragment. It is a vector of size equal to the number of resources in the machine. Each entry in the vector counts the number of cycles for which that resources would be expected to be busy as the resource requirements of a sequence of operations is accumulated within the map.

Just like for RU maps, rather than directly expose the internal structure of the mdes database to the scheduler, we provide an independent *resource minimum schedule length manager (RMSL manager)*, that is part of the mQS, and which does all the book-keeping on behalf of the scheduler. The RMSL manager is responsible for directly interfacing with the mdes database and allocating and manipulating the RMSL map as instructed by the scheduler. The RMSL manager interface queries appear in Section 6.2.4.

6 The mQS interface

This section describes the procedures that are part of the mdes query system interface. This is the recommended way of accessing mdes information from the compiler's side as shown in Figure 1. In the following specification, the description of each procedure consists of an ANSI-C style function declaration, a description of what it does, and some notes about its implementation, limitations, extensions, etc. Also, we assume that boolean values `{true, false}` and their type `bool` have been defined elsewhere.

6.1 Initializing and deallocating the mdes database

```
MDES(char *mdes_input, char *mdes_output="/dev/null", int
hashsize=1024);
```

This `MDES` class constructor reads the machine description input from the specified input file and creates the internal database. The output file specification is used for debugging. The hashsize specification controls the initial size of the opcode hashtable internal to the database. Procedures in the scheduler and register allocator interface refer directly to this database.

Note: See Section 7 for the details (format, limitations etc.) of the machine description input.

```
virtual ~MDES(void);
```

This class destructor deallocates the storage allocated to the machine description database and any other storage allocated to process the queries.

```
void set_current_MDES(MDES *mdes);
```

Installs the specified machine description database as the current default database. This procedure should be called before any of the other procedure in the mdes query interface are called with respect to the specified database.

```
void push_MDES(MDES *mdes);
```

Installs the specified machine description database as the default database saving the current database on a stack. This is used when different regions of the program need to use different mdeses. This procedure should be called before any of the other procedure in the mdes query interface are called with respect to the specified database.

```
void pop_MDES(void);
```

De-installs the current mdes database and installs the one previously saved at the top of the stack. The stack is popped. This procedure should be called after all queries to the current mdes database have been made.

6.2 Scheduler interface

Several procedures described in this section have arguments called `opcode`, `portkind`, `portnum` and `io`. Their meaning and the permissible values for them are described below.

- `opcode`: The name (a string) of a generic opcode set or a compiler-opcode.
- `portkind`: It is an enumerated value of type `IO_Portkind` that is used to identify the kind of input/output port of an operation. The possible values are given by the following *enum* declaration:

  ```
  enum IO_Portkind {DATA_IN, DATA_OUT, SYNC_IN, SYNC_OUT};
  ```
- `portnum`: It is an integer index that identifies the particular port of the operation. Predicate inputs/outputs are treated as data inputs/outputs. The predicate input is assumed to be the 0^{th} data input to an operation. The non-predicate data inputs and all data outputs start from index 1. The sync inputs/outputs start from index 0.
- `io`: This argument specifies the IO requirement of an operation in terms of an access-equivalent RS-tuple. It is a string of the form:

  ```
  [p ?] s, ...,s : d, ...,d
  ```

  `p`, `s` and `d` are names of access-equivalent register sets for the predicate, source and destination operands respectively. If the operation is not predicated, its predicate input operand specification and the associated "`?`" is dropped. Currently, each access-equivalent register set specification corresponds to either one of the physical register files of HPL-PD architecture or a literal specifier (already present in the database). For example, the IO requirement of a 2-input predicated `add` operation may look as follows:

  ```
  pr ? gpr, l : gpr
  ```

  This specifies that the predicate input comes from the register file "`pr`", the first data input comes from the register file "`gpr`", the second data input is a literal with specification "`l`", and the output goes back to the "`gpr`" register file.

6.2.1 Opcode property queries

```
int MDES_src_num(char *opcode);
```

Returns the number of source operands of the specified opcode excluding the predicate (if the opcode is predicated).

```
int MDES_dest_num(char *opcode);
```

Returns the number of destination operands of the specified opcode.

```
bool MDES_predicated(char *opcode);
```

Returns `true` if the opcode is predicated, `false` otherwise.

```
bool MDES_has_speculative_version(char *opcode);
```

Returns `true` if the opcode can be issued speculatively, `false` otherwise. The speculative opcode itself can be generated by adding a pre-specified speculative modifier to the normal opcode and is not stored within the database currently.

```
int MDES_priority(char *opcode);
```

The HPL-PD architecture permits the specification of a relative priority among the various memory ports (load/store units) [8]. This query, which is valid only for compiler-opcodes, returns the relative execution priority of that opcode. The priority guarantees a sequential order for the memory operations issued within the same cycle, if they are to the same memory location. It therefore allows the scheduler to schedule in the same instruction memory operations that might, but are expected not to, have flow, anti- and output dependences between them. This is accomplished by assigning a zero latency to the edge between such operations, and assigning a higher priority opcode to the predecessor operation than to the successor operation.

6.2.2 Latency queries

```
int MDES_init_op_io(char *opcode, char *io);
```

```
int MDES_flow_time_io(IO_portkind portkind, int portnum);
```

```
int MDES_anti_time_io(IO_portkind portkind, int portnum);
```

These three procedures are used to get various latencies associated with the inputs and outputs of a generic opcode set (compiler-opcode) in the context of a specific

IO requirement[14]. These procedures return a conservative latency summary over all alternatives present in that context. The latency descriptor associated with each alternative is used to obtain the required information.

`MDES_init_op_io`: Initializes the query process. Subsequent calls to `MDES_flow_time_io` and `MDES_anti_time_io` assume their opcode and IO requirements as specified by this call.

`MDES_flow_time_io`: If the port is a data input port, it returns the *minimum* earliest read latency, T_{er}, associated with that input across all the opcode-qualified operation sets that are consistent with the specified opcode and the IO requirement. If the port is a data output port, it returns the *maximum* latest write latency, T_{lw}, associated with that output across all the opcode-qualified operation sets. If the port is a sync input port, it returns the *minimum* earliest memory serialization latency, T_{em}, associated with that sync input across all the opcode-qualified operation sets. If the port is a sync output port, it returns the *maximum* latest memory serialization latency, T_{lm}, associated with that sync output across all the opcode-qualified operation sets. Note that these summaries are conservative in accordance with the current limitations of Elcor discussed in Section 5.1.1.

`MDES_anti_time_io`: If the port is a data input port, it returns the *maximum* latest read latency, T_{lr}, across all the opcode-qualified operation sets that are consistent with the specified opcode and the IO requirement. If the port is a data output port, it returns the *minimum* earliest write latency, T_{ew}, associated with that output across all the opcode-qualified operation sets. This query is undefined if the port is a sync input or output port. Again, these summaries are conservative.

```
int MDES_branch_latency(char *opcode);
```

Returns the branch latency, T_{br}, of the branch opcode passed as the argument. This is the latency that should be used for control dependences emanating from the branch operation. The "`opcode`" argument may be dropped in which case the latency of the opcode `BRU` is returned.

[14] As noted earlier in Section 5.3, a generic opcode set in the context of an IO requirement identifies an access-equivalent operation set.

6.2.3 Resource-usage manager (RU Manager) queries

The RU manager is responsible for directly interfacing with the machine description database and allocating and manipulating the resource usage map as instructed by the scheduler; it does not make any scheduling decisions on its own. The following queries describe this interface.

```
void RU_alloc_map(int length);

void RU_init_map(int mode, int length);

void RU_delete_map();
```

`RU_alloc_map`: Allocates the storage for a new map of size (length $\times$ the number of resources). The number of resources is taken from the machine description database. The new map is kept as a part of the internal state and subsequent calls to `RU_init_map`, `RU_delete_map` and the three scheduling procedures described later (namely, `RU_init_iterator`, `RU_get_next_nonconfl_alt` and `RU_place`) refer to this map.

`RU_init_map`: Initializes the current resource usage map. The first argument, namely mode, specifies if the resource usage map is to be used for scalar scheduling, which is denoted by mode = 0, or modulo scheduling of loops, which is denoted by mode = 1. The second argument, namely length, is the actual number of cycles in the schedule (e.g., the initiation interval, II, for modulo scheduling). Both arguments are kept as a part of the internal state and used in subsequent calls to `RU_get_next_nonconfl_alt` and `RU_place`.

`RU_delete_map`: Deallocates the storage for the map.

A map of sufficient length, once allocated, can be used for scheduling more than one region or scheduling a loop at more than one II by repeatedly calling `RU_init_map`.

```
void RU_init_iterator(char *opcode, void *op, char *io, int time);

bool RU_get_next_nonconfl_alt(char **opcode, int *priority);

void RU_place();
void RU_get_conflicting_ops(Hash_set<void*>& ops);
```

These procedures are used to schedule operations. Given an access-equivalent operation set and a candidate time at which the scheduler is trying to place the operation, we have to find an opcode-qualified operation set whose resource requirements don't conflict with the current resource usage map. If such an opcode-qualified operation set is found, then the operation can be scheduled using that opcode-qualified operation set.

The arguments passed to `RU_init_iterator` are kept as part of the internal state and used by subsequent calls to `RU_get_next_nonconfl_alt` and `RU_place`. The state also keeps a pointer to the reservation table of the non-conflicting opcode-qualified operation set found by the last call to `RU_get_next_nonconfl_alt`. This pointer is used for two purposes: either to place the operation using the opcode-qualified operation set by calling `RU_place` or to continue the search for next non-conflicting opcode-qualified operation set. The initial value of this internal pointer is undefined.

`RU_init_iterator`: Initializes the search process for an access-equivalent operation set as given by the "`opcode`" and the IO requirement "`io`". The argument "`op`" is a pointer to the current operation being scheduled within the computation graph in order to facilitate backtracking. The argument "`time`" specifies the time (relative to the beginning of the schedule) at which the scheduler is trying to schedule the operation.

`RU_get_next_nonconfl_alt`: This function may be called repeatedly to search for the next available non-conflicting opcode-qualified operation set for the current access-equivalent operation set. The arguments are simply empty spaces for the results. The search is carried out by scanning through each of the (remaining) alternative descriptors present under the current access-equivalent operation set and matching its reservation table offset by the current scheduling time with the remaining available slots on the resource-usage map. If it finds an alternative that does not conflict, it returns `true`. In addition, it sets the first argument to the name of the corresponding compiler-opcode and the second argument to the relative execution priority for the functional unit corresponding to that compiler-opcode. If the procedure finds no non-conflicting alternative, it returns `false` and sets both the arguments to unspecified values.

If the mode is modulo scheduling (see `RU_init_map`), all timing calculation are done modulo II.

`RU_place`: Places (commits) a node in the schedule. It modifies the current resource-usage map to indicate that the access-equivalent operation set has been scheduled at the given time using the reservation table of the alternative found by the last call to `RU_get_next_nonconfl_alt`. (Note that the internal state contains all the information needed by this procedure).

If the mode is modulo scheduling (see `RU_init_map`), all timing calculation are done modulo II.

`RU_get_conflicting_ops`: If the scheduler needs to backtrack, this function returns all the previously scheduled operations that conflict with any alternative of the current access-equivalent operation set if scheduled at the current cycle. The idea is that if all these operations were to be unscheduled, the scheduler would then have complete freedom in scheduling the current operation.

```
void RU_print_map(FILE *file_pointer);
```

It prints the resource usage map. The argument is a pointer to an open file where the output is to be sent. The main use of this procedure is in debugging the compiler.

```
void RU_remove(void *op, char *io, int time);
```

It de-schedules and removes the given opcode-qualified operation set, previously scheduled at the given time, from the internal resource-usage map.

6.2.4 Resource Minimum Schedule Length (RMSL) manager queries

```
void RMSL_alloc();
```

```
void RMSL_dealloc();
```

```
void RMSL_init();
```

These functions allocate, deallocate and initialize the internal resource counters for the computation of the resource lower bound.

```
void RMSL_nextop(char *opcode, char *io);
```

Accumulates the resource counts for the best alternative from the given access-equivalent operation set (as specified by the opcode and the IO requirement). It selects the least resource-critical alternative based on the current accumulation, i.e., the alternative that advances the maximum resource count across all resources by the minimum amount. In this computation, the times of the resource usages are ignored; only the specific resources used are counted.

```
int RMSL_value();
```

Returns the computed value of RMSL as the maximum resource usage count over all resources. The returned value takes into account all the operations submitted since the last call to `RMSL_init`.

The following points should be noted:

- `RMSL_value` doesn't re-initialize the internal data structures.
- It is possible to get the value of RMSL incrementally. That is, it is possible to submit a set of operations, call `RMSL_value` to get the value of RMSL taking these operations into account, submit an additional set of operations and then call `RMSL_value` again to get the value of RMSL taking the operations in both the sets into account.
- These procedures implement a specific heuristic for calculating RMSL. The reason is that an opcode may have multiple reservation tables, and the calculation of RMSL involves choosing a specific reservation table. The current implementation uses the greedy approach. It is possible to implement more complex heuristics or even a full branch and bound algorithm by keeping all the operations in the internal data structures and computing the value of RMSL only when the procedure `RMSL_value` is called.

6.3 Register allocator interface

Several procedures described below take the name of a physical register file as an argument `regname`. This name should the one of those specified in the machine description database.

```
void MDES_reg_names(List<char*>& regnames);
```

Fills the given list with the names of all the distinct access-equivalent register sets supported by the current mdes. For example, single-precision floating-point registers and their odd-even pairs constituting double-precision floating-point registers would be considered as distinct compiler-register sets that have structural overlap. Currently, these sets correspond directly to the physical register files in the HPL-PD architecture.

```
int MDES_reg_width(char *regname);
```

Returns the width of the given register set. All registers within the same register set are assumed to be of the same width.

```
int MDES_reg_static_size(char *regname);
```

Returns the number of static registers in the given register set.

```
int MDES_reg_rotating_size(char *regname);
```

Returns the number of rotating registers in the given register set.

```
int MDES_supports_rot_reg(char *regname);
```

Returns whether the given register set supports rotating register addressing or not.

```
int MDES_reg_has_speculative_bit(char *regname);
```

Returns whether the given register set has an extra tag bit to support speculative execution.

```
int MDES_reg_is_allocatable(char *regname);
```

Returns whether the given register set can be considered for register allocation. For instance, literal registers are not allocatable since they contain a fixed value.

7 External file format for the machine-description database

In order to specify the machine description information externally, the Elcor compiler currently uses DBL, a general relational database description language [20]. DBL supports a high-level human-editable textual form and a low-level machine-readable form. Tools are provided to expand macros in the high-level form and compile it down to the low-level form.

The exact definitions of the various database relations used by a particular database are defined within the same DBL framework and serve to specialize the file format for a given database schema. For instance, the high-level machine description database specification currently used within Elcor is called Hmdes Version 2 or Hmdes2 [9] and is defined on top of DBL. The machine description for a given target architecture is expressed in Hmdes2 format as a text file. After macro processing and compilation, the corrresponding low-level specification, expressed in Lmdes Version 2, is loaded into Elcor using the `MDES` class constructor described previously. In this section, we briefly describe the various sections of the Hmdes2 specification for Elcor. We assume that the reader is familiar with the basic DBL structure and syntax [9].

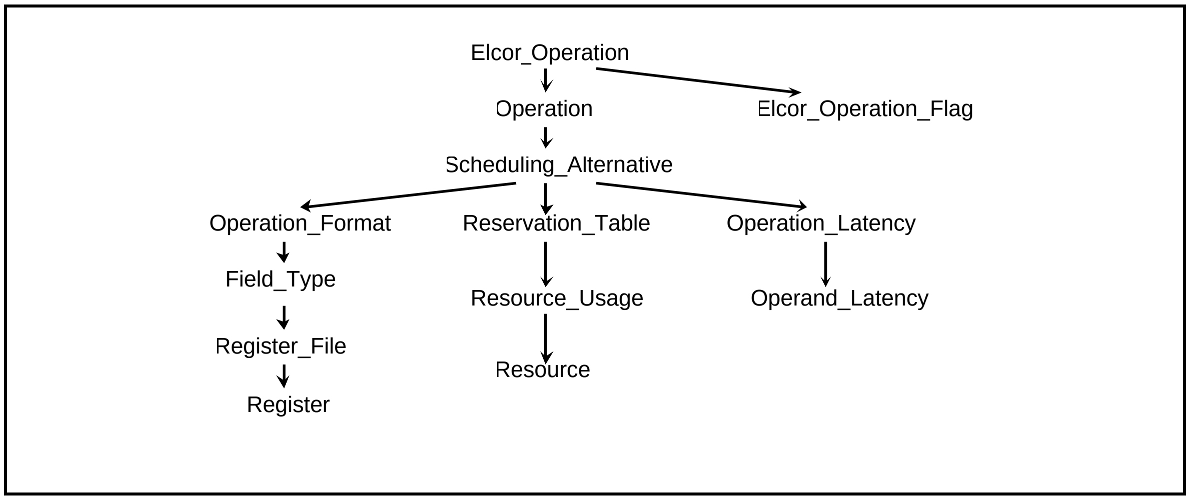


Figure 12. The Hmdes2 section hierarchy

Figure 12 shows the hierarchy of sections defined within Hmdes2 database file format. For each of the sections in bottom-up order (as it should appear in the Hmdes2 file), we describe below the schema and show an example. All the schema declarations are already defined in a separate file called `structure_pristine.hmdes2` which may simply be included into the final Hmdes2 file.

7.1 Register section

Schema:

```
CREATE SECTION Register
  OPTIONAL overlaps(LINK(Register)*); // list of overlapping compiler-registers
{}
```

Example:

```
SECTION Register {
  GPR0(); GPR1(); … GPR63();            // simple compiler-registers
  'GPR[0]'(); 'GPR[1]'(); … 'GPR[63]'();
  ...
  CR0(overlaps(PR0 … PR31));            // aliased compiler-registers
  ...
}
```

The register section defines all the compiler-registers of the machine and their mutual overlap. For instance, both registers from an odd-even pair of single precision floating-point registers are considered to overlap the corresponding double precision register which is defined separately.

Note: This section corresponds to the register descriptors in the mdes. Since they are missing in the current implementation of Elcor, this section is currently ignored.

7.2 Register file section

Schema:

```
CREATE SECTION Register_File
  REQUIRED width(INT);                  // width of each register in the file
  OPTIONAL virtual(STRING);             // name of generic register set
  OPTIONAL static(LINK(Register)*);     // number of static registers
  OPTIONAL rotating(LINK(Register)*);// number of rotating registers
  OPTIONAL speculative(INT);            // whether spec. tag bit is present
  OPTIONAL intlist(INT*);               // list of integer literals (for literal file)
  OPTIONAL intrange(INT*);              // range of integer literals (for literal file)
  OPTIONAL doublelist(DOUBLE*);        // list of floating pt literals (for literal file)
{}
```

Example:

```
SECTION Register_File {
  RF_i(width(32) virtual(i) speculative(1)    // integer register file
      static(GPR0 … GPR63) rotating('GPR[0]' … 'GPR[63]'));
  LF_s(width(6) virtual(l) intrange(-32 31)); // literals -32 to 31
  ...
  LF_l(width(32) virtual(l));                 // general literal file
  RF_u(width(0) virtual(u));                  // bit bucket
}
```

The register file section defines the access-equivalent register sets of the machine. Currently this directly corresponds to the architectural register files of the HPL-PD architecture. The specified properties include the name of the generic register set that these compiler-registers belong to, the width of each of the registers, the list of static and rotating registers, and whether or not there is an extra speculative tag bit with each register in the set (0=non-speculative, 1=speculative). If this is a literal file then additional fields may be used to specify either a list of actual integer or double precision literals implemented, or a set of integer ranges expressed as pairwise list of inclusive lower and upper bounds.

7.3 Field type (IO set) section

Schema:

```
CREATE SECTION Field_Type
  OPTIONAL regfile(LINK(Register_File));        // single acc. eq. register set
  OPTIONAL compatible_with(LINK(Field_Type)*); // set of acc. eq. register sets
{}
```

Example:

```
SECTION Field_Type {

  FT_i(regfile(RF_i));

  FT_c(regfile(RF_c));

  FT_l(regfile(LF_l));

  ...

  FT_icl(compatible_with(FT_i FT_c FT_l));

  ...

}
```

Entries in the field type section identify singleton or collections of access-equivalent register sets that may be used as the source or destination of various operations.

7.4 Operation Format (IO descriptor) section

Schema:

```
CREATE SECTION Operation_Format

  OPTIONAL pred(LINK(Field_Type)*);  // guarding predicate

  OPTIONAL src(LINK(Field_Type)*);   // source operands

  OPTIONAL dest(LINK(Field_Type)*);  // destination operands

{}
```

Example:

```
SECTION Operation_Format {

  OF_intarith2(pred(FT_p) src(FT_icl FT_icl)

                dest(FT_ic));

  OF_intcmppred(pred(FT_p) src(FT_il FT_il)
                dest(FT_p FT_p));

  ...

}
```

Each entry in the operation format section defines an IO descriptor which represents a set of access-equivalent RS-tuples in a compact way. An entry consists of the field type (IO set) for the predicate operand port (if any) and zero or more field types for the source and destination operand ports. This also indirectly identifies the number of source and destination operand ports as well as whether the operation (which refers to this IO descriptor) is predicated or not.

A word of caution: If the IO descriptors associated with an opcode (set) are not disjoint, the scheduler queries will just select the first IO descriptor that it encounters in the IO_list that matches the IO requirement. This is not necessarily the first one listed in the Hmdes2 file[15].

7.5 Resource section

Schema:

```
CREATE SECTION Resource          // any hardware or abstract resource

{}
```

Example:

```
SECTION Resource {

  i0(); i1(); f0(); m0(); b0(); // 2 int, 1 float, 1 mem, and 1 branch units

}
```

The resource section defines the various physical or logical resources of the machine. For instance, an integer unit, a hardware pipeline stage, a result bus, or an instruction field in the instruction, that is currently being decoded, are all examples of resources. It is up to the

[15] In the current implementation, we have tried to preserve the order of appearance of various scheduling alternatives in the external specification (Section 7.10) as the internal order of the IO list elements and the alternatives associated with each IO descriptor. However, relying upon this ordering is not recommended.

mdes writer to decide in what detail he or she wants to describe the machine and the resource usage of its operations.

7.6 Resource usage section

Schema:

```
CREATE SECTION Resource_Usage
  REQUIRED use(LINK(Resource));     // resource to be used
  REQUIRED time(INT INT*);          // at time(s)
{}
```

Example:

```
SECTION Resource_Usage {
  RU_i0(use(i0) time(0));           // use int unit 0 at time 0
  RU_i1(use(i1) time(0));           // use int unit 1 at time 0
  ...
}
```

The resource usage section defines all possible resource-time pairs that are used within any reservation table present within the database. The syntax allows a factored representation of the pairs when the same resource is used at multiple times.

7.7 Resource unit, table option, and reservation table sections

Schema:

```
CREATE SECTION Resource_Unit                // AND-style resource usages
  REQUIRED use(LINK(Resource_Usage) LINK(Resource_Usage)*);
{}
CREATE SECTION Table_Option                 // OR-style resource usages
```

```
  REQUIRED one_of(LINK(Resource_Unit|Resource_Usage)
                LINK(Resource_Unit|Resource_Usage)*);
{}
CREATE SECTION Reservation_Table
  REQUIRED use(LINK(Resource_Usage|Resource_Unit|Table_Option)*);
{}
```

Example:

```
SECTION Reservation_Table {
  RT_null(use());         // null reservation table for dummy ops
  RT_i0(use(RU_i0));      // simple table with int unit 0 being used at time 0
  RT_i1(use(RU_i1));
  ...
}
```

The reservation table section declares all the possible reservation patterns of the various opcode-qualified operation sets and fully qualified operations present within the database. Each reservation table is capable of representing a complex AND-OR pattern of resource usages through the use of `Table_Option` and `Resource_Unit` sections. A *resource unit* is a group of AND-style resource usages that are logically used together, e.g., an integer resource unit may consist of an instruction slot resource, two input bus resources sourcing from the integer register file, and an integer ALU resource being used at time 0, and a result bus back to the integer register file being used at time 1. Grouping these resource usages together in a unit enables efficient representation and query within the mQS [21]. Similarly, a *table option* is a group of OR-style resource units or resource usages, any one of which may be used by a particular operation, e.g., in a machine with two integer pipelines that are not necessarily identical, an integer operation may be issued on either one of them[16].

[16] In case of multiple identical resources a counted representation may result in even more compact implementation, but for a dissimilar set of resource choices an OR-style table option is needed.

The current implementation, however, supports only a simple AND-style reservation pattern as a list of resource usages directly contained within a reservation table. Therefore the `Table_Option` and `Resource_Unit` sections are currently ignored.

7.8 Operand latency section

Schema:

```
CREATE SECTION Operand_Latency
  REQUIRED time(INT*);          // (possibly multiple) timings for a single operand
{}
```

Example:

```
SECTION Operand_Latency {
  time_null(time(0));           // always zero, for dummy ops
  time_int_alu_sample(time(0));
  time_int_alu_exception(time(0));
  time_int_alu_latency(time(1));
  time_int_alu_reserve(time(0));
  ...
}
```

The operand latency section identifies all possible latency times used within the database.

7.9 Operation latency section

Schema:

```
CREATE SECTION Operation_Latency
  OPTIONAL dest(LINK(Operand_Latency)*);        // T_lw
  OPTIONAL src(LINK(Operand_Latency)*);         // T_er
  OPTIONAL pred(LINK(Operand_Latency)*);        // $T_{er}$
  OPTIONAL exc(LINK(Operand_Latency));          // $T_{lr}$ (common)
  OPTIONAL rsv(LINK(Operand_Latency)*);         // $T_{ew}$
  OPTIONAL sync_dest(LINK(Operand_Latency)*);   // $T_{lm}$ or $T_{br}$
  OPTIONAL sync_src(LINK(Operand_Latency)*);    // $T_{em}$
  OPTIONAL mem_dest(LINK(Operand_Latency)*);    // unused
  OPTIONAL mem_src(LINK(Operand_Latency)*);     // unused
  OPTIONAL ctrl_dest(LINK(Operand_Latency)*);   // unused
  OPTIONAL ctrl_src(LINK(Operand_Latency)*);    // unused
{}
```

Example:

```
SECTION Operation_Latency {
  OL_int(dest(time_int_alu_latency … time_int_alu_latency)
         src(time_int_alu_sample … time_int_alu_sample)
         pred(time_int_alu_sample)
         exc(time_int_alu_exception)
         rsv(time_int_alu_reserve … time_int_alu_reserve)
         sync_dest(time_int_alu_sample time_int_alu_sample)
         sync_src(time_int_alu_sample time_int_alu_sample));
}
```

Each entry in the operation latency section defines all the latency parameters of the various operands of a schedule-equivalent operation set. The `pred/src` and `dest` latencies correspond to T_{er} and T_{lw}, respectively, one each per input or output. The `exc` latency records a single common T_{lr} for all of the input operands[17]. The `rsv` latencies correspond to T_{ew}, one for each output[18]. When computing delays on Memory edges (which are hooked up to the Mem ports of load/store operations), the first `sync_src` and first `sync_dest` latencies of the load/store operation are used to specify T_{em} and T_{lm}, respectively. When computing delays on Control1 edges (which are hooked up to the Control1 ports of operations), the first `sync_dest` latency is used to specify T_{br}[19]. The `mem_src`, `mem_dest`, `ctrl_src` and `ctrl_dest` latencies are currently unused.

7.10 Scheduling alternative section

Schema:

```
CREATE SECTION Scheduling_Alternative

  REQUIRED format(LINK(Operation_Format)LINK(Operation_Format)*);

  REQUIRED resv(LINK(Reservation_Table));

  REQUIRED latency(LINK(Operation_Latency));

{}
```

Example:

```
SECTION Scheduling_Alternative {

  SA_intarith2_int_i0(format(OF_intarith2)

                    resv(RT_i0) latency(OL_int));   // integer unit 0

  SA_intarith2_int_i1(format(OF_intarith2)

                    resv(RT_i1) latency(OL_int));   // integer unit 1

  ...
```

[17] This is a restriction of the current Hmdes2 schema and the mdes data structures as discussed in Section 5.1.1. Typically, this is set to the longest latency with which an exception can be reported.

[18] At the time of writing this document, this latency is set to the conservative value of 0 for every output. This is a result of our previous conceptualization of the exception handling problem.

[19] Note that, currently, this is not provided globally for all branches.

```
}
```

A scheduling alternative is a triple consisting of one or more IO descriptors, a reservation table and a latency descriptor that are common among a set of opcode-qualified operation sets possibly even belonging to different generic operation sets. Note that a scheduling alternative is not the same as an alternative. The latter is the triple consisting of a compiler-opcode, a reservation table and an operation latency descriptor and it encodes information specific to a single opcode-qualified operation set.

The scheduling alternative section is used merely as a syntactic device for sharing common information within the external format; the Hmdes2 customizer distributes this information appropriately across the internal data-structures upon reading the external specification.

7.11 Operation (opcode-qualified) section

Schema:

```
CREATE SECTION Operation

  REQUIRED alt(LINK(Scheduling_Alternative)*);

{}
```

Example:

```
SECTION Operation {                     // operation sets with compiler-opcode

  'addw.0'(alt(SA_intarith2_int_i0));  // addw.0 on integer unit 0

  'addw.1'(alt(SA_intarith2_int_i1));  // addw.1 on integer unit 1

  ...

  'dummy_branch.0'(alt(SA_dummy_null_null));

  'control_merge.0'(alt(SA_dummy_null_null));

  ...

}
```

The operation section defines all the opcode-qualified operation sets present within the database. Each entry represents (possibly) a set of opcode-qualified operation sets sharing the same compiler-opcode and characterized by one or more access-equivalent RS-tuples as implied by the IO descriptors present within the scheduling alternative.

7.12 Elcor operation flag section

Schema:

```
CREATE SECTION Elcor_Operation_Flag

{}
```

Example:

```
SECTION Elcor_Operation_flag {

  NULL(); INT(); FLOAT(); MEMORY(); BRANCH();  // various resource classes

  NOSPEC(); SPEC();                            // speculative or not

  ...

}
```

The Elcor operation flag section contains compiler flags that may be used to denote useful compiler properties of operations such as the resource class of an operation, whether it can be issued speculatively or not etc. We assume that if an access-equivalent operation set can be executed speculatively, then all opcode-qualified operation sets, to which it maps, can also be executed speculatively.

7.13 Elcor operation (generic) section

Schema:

```
CREATE SECTION Elcor_Operation

  OPTIONAL op(LINK(Operation)*);

  OPTIONAL flags(LINK(Elcor_Operation_Flag)*);

{}
```

Example:

```
SECTION Elcor_Operation {

  ADD_W(op('addw.0' 'addw.1') flags(INT SPEC));

  ...

  dummy_branch(op('dummy_branch.0')

               flags(NULL NOSPEC));

  control_merge(op('control_merge.0')

                 flags(NULL NOSPEC));

  ...

}
```

The Elcor operation section defines all generic operation sets of the machine. For the HPL-PD architecture, these are also the access-equivalent operation sets. Each entry simply enumerates all the opcode-qualified operation sets present within the set along with the flags shared by all operations in the set.

8 Related work

8.1 Antecedents

In the early 1980's, mdes-driven compilers [22, 23] were developed for the VLIW mini-supercomputer products that were built and marketed by Cydrome [6] and Multiflow [5]. The work discussed in this report is a direct descendant of the Cydrome work on mdes-driven compilers which was motivated by the very pragmatic need to be able to develop a compiler while the details of the machine were still in flux. The machine modelling capability had to be quite sophisticated since the Cydra 5 was a real product for which design decisions had to be made which violated many of the simplifying assumptions often made by researchers (such as single cycle operations, fully pipelined functional units, simple reservation tables and homogeneous, even identical, functional units).

This work was continued in 1989 as part of the EPIC research at HP Laboratories by the authors, but with the motivation of being able to experiment with and evaluate widely varying EPIC processors. Subsequently, in collaboration with the University of Illinois' IMPACT project, the mdes-driven capability was further developed with an eye to supporting certain aspects of superscalar processors. Additionally, a high level, human-friendly machine description language was developed to facilitate the definition of the mdes for a machine [20, 9]. HP Laboratories' Elcor compiler and the University of Illinois' IMPACT compiler are both mdes-driven as far as scheduling and register allocation are concerned.

8.2 Other work

An interesting sub-topic of mdes-driven code generation for EPIC processors is that of machine description optimization. Both scheduling time and mdes size can be decreased if the reservation tables are represented as AND/OR trees of reservation table fragments [21]. This increases the amount of factorization of common pieces of reservation tables and reduces the size of the database by reusing this information. A complementary optimization is to construct a simpler set of synthetic reservation tables corresponding to a smaller number of synthetic resources but which are equivalent to the original reservation tables with respect to the constraints that they place upon the legal relative schedule times of every pair of operations [24].

Acknowledgements

Many of the concepts discussed in this report have been strongly influenced by, and have greatly benefited from, discussions with Mike Schlansker over the past ten years. We are indebted to John Gyllenhaal for his design of the DBL data description language and the Hmdes2 machine-description schema, as well as for the software that processes and translates them down to a machine-readable format.

References

1. B. R. Rau, C. D. Glaeser and E. M. Greenawalt. Architectural support for the efficient generation of code for horizontal architectures. Proc. Symposium on Architectural Support for Programming Languages and Operating Systems (Palo Alto, March 1982), 96-99.

2. J. A. Fisher. Very long instruction word architectures and the ELI-512. Proc. Tenth Annual International Symposium on Computer Architecture (Stockholm, Sweden, June 1983), 140-150.

3. TMS320C62xx CPU and Instruction Set Reference Guide. (Texas Instruments, 1997).

4. Trimedia TM-1 Media Processor Data Book. (Philips Semiconductors, Trimedia Product Group, 1997).

5. R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth and P. K. Rodman. A VLIW architecture for a trace scheduling compiler. IEEE Transactions on Computers C-37, 8 (August 1988), 967-979.

6. B. R. Rau, D. W. L. Yen, W. Yen and R. A. Towle. The Cydra 5 departmental supercomputer: design philosophies, decisions and trade-offs. Computer 22, 1 (January 1989), 12-35.

7. M. Schlansker, B. R. Rau, S. Mahlke, V. Kathail, R. Johnson, S. Anik and S. G. Abraham. Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity. HPL Technical Report HPL-96-120. Hewlett-Packard Laboratories, February 1997.

8. V. Kathail, M. Schlansker and B. R. Rau. HPL-PD Architecture Specification: Version 1.1. Technical Report HPL-93-80 (R.1). Hewlett-Packard Laboratories, September 1998.

9. J. C. Gyllenhaal, W.-m. W. Hwu and B. R. Rau. HMDES Version 2.0 Specification. Technical Report IMPACT-96-3. University of Illinois at Urbana-Champaign, 1996.

10. B. R. Rau, V. Kathail and S. Aditya. Machine-Description Driven Compilers for EPIC Processors. HPL Technical Report HPL-98-40. Hewlett-Packard Laboratories, September 1998.

11. B. R. Rau. Data flow and dependence analysis for instruction level parallelism, in Fourth International Workshop on Languages and Compilers for Parallel Computing, U. Banerjee, D. Gelernter, A. Nicolau and D. Padua (Editor). (Springer-Verlag, 1992), 236-250.

12. B. R. Rau, M. S. Schlansker and P. P. Tirumalai. Code generation schemas for modulo scheduled loops. Proc. 25th Annual International Symposium on Microarchitecture (Portland, Oregon, December 1992), 158-169.

13. G. M. Silberman and K. Ebcioglu. An architectural framework for supporting heterogeneous instruction-set architectures. Computer 26, 6 (June 1993), 39-56.

14. S. A. Mahlke, W. Y. Chen, R. A. Bringmann, R. E. Hank, W. W. Hwu, B. R. Rau and M. S. Schlansker. Sentinel scheduling: a model for compiler-controlled speculative execution. ACM Transactions on Computer Systems 11, 4 (November 1993), 376-408.

15. E. S. Davidson, L. E. Shar, A. T. Thomas and J. H. Patel. Effective control for pipelined computers. Proc. COMPCON '90 (San Francisco, February 1975), 181-184.

16. M. S. Schlansker and V. Kathail. Critical path reduction for scalar programs. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995), 57-69.

17. A. E. Eichenberger and E. S. Davidson. Register allocation for predicated code. Proc. 28th Annual International Symposium on Microarchitecture (Ann Arbor, Michigan, November 1995), 180-191.

18. D. M. Gillies, D.-c. R. Ju, R. Johnson and M. Schlansker. Global predicate analysis and its application to register allocation. Proc. 29th Annual IEEE/ACM International Symposium on Microarchitecture (Paris, France, December 1996), 114-125.

19. B. R. Rau. Iterative modulo scheduling. International Journal of Parallel Processing 24, 1 (February 1996), 3-64.

20. J. C. Gyllenhaal. A Machine Description Language for Compilation. M. S. Thesis. Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign, 1994.

21. J. C. Gyllenhaal, W.-m. W. Hwu and B. R. Rau. Optimization of machine descriptions for efficient use. Proc. 29th Annual IEEE/ACM International Symposium on Microarchitecture (Paris, France, December 1996), 349-358.

22. J. C. Dehnert and R. A. Towle. Compiling for the Cydra 5. The Journal of Supercomputing 7, 1/2 (May 1993), 181-228.

23. P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell and J. C. Ruttenberg. The Multiflow trace scheduling compiler. The Journal of Supercomputing 7, 1/2 (May 1993), 51-142.

24. A. E. Eichenberger and E. S. Davidson. A reduced multipipeline machine description that preserves scheduling constraints. Proc. SIGPLAN'96 Conference on Programming Language Design and Implementation (Philadelphia, Pennsylvania, May 1996), 12-20.